University of Alberta

Library Release Form

Name of Author: Justin S.N. Gamble

Title of Thesis: Towards a Methodology for Intelligent Activity Monitoring

Degree: Master of Science

Year this Degree Granted: 2001

There are two ways of constructing a software design; one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.

C. A. R. Hoare

University of Alberta

Towards a Methodology for Intelligent Activity Monitoring

by

Justin S.N. Gamble

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Fall 2001

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research for acceptance, a thesis entitled **Toward a Methodology for Intelligent Activity Monitoring** submitted by Justin S.N. Gamble in partial fulfillment of the requirements for the degree of **Master of Science**.

To my mother Suzanne,
for her support and patience

Abstract

In many of today's industries, equipment pieces are monitored and values are recorded to reflect the productivity and health of each machine. For example: input rate, output rate, machine temperature, speed of operation, amount of current being drawn, ...etc. This continuous data gathering results in a wealth of data, but the transformation of this data into useful knowledge remains a tremendous challenge.

A critical missing piece is information about the activity that the equipment is engaged in while the data gathering is occurring. This contextual information helps us analyse and understand the accumulated data, allowing for better equipment operation business decisions.

This thesis focuses on *Intelligent Activity Monitoring* (IAM), which we define as the use of an automatic solution to track the behaviour of physical equipment. We propose applying a subset of Statecharts towards the modelling of industrial activities. Statecharts are a visual formalism frequently used for detailing the internal behaviour of software programs. A prototype has been built which allows the execution of Statechart specifications. This prototype is implemented in the event-driven SICLE language, which we extended for our application purpose. We apply our approach and tool towards the monitoring of one of Syncrude's oil sand crushers during its winter operation.

Acknowledgements

I would like to express my deep appreciation to my supervisor, Dr. Pawel Gburzynski, for his support, encouragement, and helping me to deal with many important matters.

I am indebted to Dr. Ron Kube, Senior Research Scientist at Syncrude, without whom this work would not have come about. The thesis topic originated with him, and I am grateful for his guidance and enthusiam.

Thanks are due to Dr. Yannis Nikolaidis for lending an open ear to my difficulties, for lending numerous reading materials, for introducing me to MySQL, and for proof-reading early drafts of this thesis.

I am grateful to Lloyd Goethals for providing input data from the Syncrude tape archives, and Lionel Cayer for answering my detailed questions on the crusher performance.

Contents

1 Introduction **1**
- 1.1 Problem definition . . . 1
- 1.2 Syncrude partnership . . . 1
- 1.3 Motivation . . . 2
- 1.4 Purpose . . . 4
- 1.5 Goal . . . 4
- 1.6 Overview . . . 5

2 Related Work **6**
- 2.1 Monitoring systems . . . 6
- 2.2 Visually modelling industrial environments . . . 7
  - 2.2.1 Petri Nets . . . 7
    - 2.2.1.1 Overview . . . 7
    - 2.2.1.2 Modelling examples . . . 10
  - 2.2.2 Statecharts . . . 12
    - 2.2.2.1 Overview . . . 12
    - 2.2.2.2 Modelling examples . . . 14
    - 2.2.2.3 Implementations of Statecharts . . . 15
    - 2.2.2.4 Semantics . . . 16
- 2.3 Discussion . . . 18
  - 2.3.1 Modelling language . . . 18
  - 2.3.2 Purpose . . . 18
  - 2.3.3 Modelling the world . . . 19

3 Activity Programs **20**
- 3.1 Definitions and terminology . . . 20
  - 3.1.1 States . . . 20
  - 3.1.2 Events . . . 21
  - 3.1.3 Transitions . . . 21
  - 3.1.4 Guards . . . 21
- 3.2 Activity Model diagrams . . . 22
  - 3.2.1 Basic states . . . 22
  - 3.2.2 State transitions . . . 22
  - 3.2.3 Cluster states . . . 23
  - 3.2.4 Orthogonal states . . . 24
  - 3.2.5 Special transitions . . . 25
    - 3.2.5.1 Guard branches . . . 27
    - 3.2.5.2 Junction connectors . . . 28
    - 3.2.5.3 Diagram connectors . . . 29
- 3.3 Program structure . . . 29
  - 3.3.1 The transfer from input to output . . . 30
  - 3.3.2 The internals of a client . . . 30

3.3.2.1 Soft sensors . . . 32
3.3.2.2 Central blackboard . . . 32
3.3.2.3 Knowledge sources . . . 33
3.4 Event specification . . . 33
3.5 Guard specification . . . 35
3.6 Response actions . . . 36
3.7 Mapping the model to a program . . . 38
3.7.1 Format of an Activity Program . . . 38
3.7.2 Specifying input . . . 41
3.7.3 Mapping events and guards . . . 41
3.7.4 Adding procedures . . . 42
3.7.5 Mapping states . . . 42
3.7.6 Mapping response actions . . . 43
3.7.7 Mapping transitions . . . 43
3.8 Example: truck torsion tubes . . . 45
3.8.1 Motivation for monitoring trucks . . . 45
3.8.2 Model design . . . 46
3.8.3 Model review . . . 48
3.9 Summary . . . 50

4 Experiment and Results **51**
4.1 Problem setup . . . 51
4.1.1 Syncrude crushers in the winter . . . 51
4.1.2 Purpose of study . . . 53
4.1.2.1 Downtime activities . . . 54
4.1.2.2 Crusher utilization . . . 55
4.1.3 Input data . . . 55
4.1.4 Scheduled maintenance . . . 56
4.2 Solution design . . . 56
4.2.1 Partial IAM model . . . 56
4.2.2 IAM model . . . 58
4.2.2.1 The states . . . 58
4.2.2.2 The transitions . . . 61
4.2.2.3 The thresholds . . . 62
4.3 Results and discussion . . . 63
4.4 Further work . . . 69
4.5 Summary . . . 69

5 Looking under the hood **71**
5.1 Introduction to SICLE . . . 71
5.1.1 Object-oriented . . . 71
5.1.2 Entities . . . 72
5.1.3 Transition commands . . . 74
5.1.4 Event interruptions . . . 75
5.1.5 Communication . . . 75
5.1.5.1 Internal . . . 75
5.1.5.2 External . . . 77
5.2 Implementing IAM with SICLE . . . 77
5.2.1 Passing information between modes . . . 78
5.2.2 Short-term event memory . . . 78
5.2.3 IAM states . . . 78
5.2.3.1 Mailbox hierarchy . . . 79
5.2.3.2 Common properties . . . 81
5.2.3.3 Basic states . . . 81

5.2.3.4 Cluster states . . . 84
5.2.3.5 Orthogonal states . . . 86
5.2.4 IAM transitions . . . 87
5.2.4.1 Events . . . 87
5.2.4.2 Guards . . . 89
5.2.5 DRS implementation . . . 89
5.2.6 Communicating with the DRS . . . 90
5.2.6.1 Protocol . . . 90
5.2.6.2 Dealing with fast input . . . 90
5.2.6.3 Scheduling . . . 91
5.3 Database storage . . . 92
5.3.1 ER diagram . . . 92
5.3.2 Database tables . . . 92
5.4 Summary . . . 92

6 Conclusion **95**
6.1 Summary . . . 95
6.2 Contribution . . . 95
6.3 Limitations . . . 96
6.4 Possibilities for further research . . . 96
6.4.1 Graphical user interface . . . 96
6.4.2 Model validation . . . 96
6.4.3 Machine learning . . . 97
6.4.4 Real-time execution . . . 97
6.4.5 Long-term direction: condition-based maintenance . . . 97

A Source Code **99**
A.1 Truck Model . . . 99
A.2 Crusher Model . . . 102

Bibliography **107**

List of Figures

1.1 Caterpillar 793B torsion tube torque measurement 2

2.1 Firing a Petri Net transition 8
2.2 Petri Net with an inhibitor arc 9
2.3 A coloured Petri Net 10
2.4 Hierarchical Petri Net 11
2.5 Statecharts Example 13
2.6 Exit-safe sub-states 16

3.1 A state without sub-states 22
3.2 Format of a transition between states 22
3.3 The format of a cluster 23
3.4 The same diagram drawn with and without clusters 24
3.5 Methods of entering and exiting an orthogonal state 26
3.6 The same diagram drawn with and without orthogonal states 26
3.7 A comparison with and without a guard branch 27
3.8 A comparison with and without a junction connector 28
3.9 Transferring to far away states 29
3.10 Program structure 31
3.11 Specifying states using text 42
3.12 Caterpillar 793B haul truck being loaded by a Demag shovel 45
3.13 First design 46
3.14 Second design 47
3.15 Final design 49

4.1 Arrangement of crusher components 52
4.2 Plugged crusher 53
4.3 An operation interruption caused by a frozen oil sand lump 54
4.4 Key states of crusher model 57
4.5 Crusher model 59
4.6 Histogram of Roll Speed values 64
4.7 Total time within each state, January 2000 66
4.8 Total time within each state, February 2000 67
4.9 Summary of activity durations within CrusherDown state 68

5.1 Hierarchical breakdown of an activity model 80
5.2 Modes of an IAM basic state 82
5.3 Example of a merge transition 84
5.4 Example of a cluster and sub-state each pointing to different next states 86
5.5 Protocol between DRS and IAM 90
5.6 Entity-Relationship model of database organization 93
5.7 Database tables 94

List of Tables

3.1 Supported event types 34
3.2 Guard syntax written in Backus-Naur form (BNF) 36
3.3 Functions calls useful within response actions 37
3.4 Commands used within Activity Programs 40

4.1 Input sources of crusher data 56
4.2 Events and guards used in the crusher model 58
4.3 Crusher model states 60
4.4 Crusher model transitions 61
4.5 Time spent in each state 65

5.1 Object-oriented commands 73
5.2 SICLE's transition commands 75
5.3 Supported AIs and their events 76
5.4 SICLE inter-process communication commands 77
5.5 Commands unique to SICLE 78
5.6 Modes of a basic state 83
5.7 Supplementary SICLE command: ignoreStates 86
5.8 Implementation of IAM event types 87

Chapter 1

Introduction

1.1 Problem definition

In many of today's industries, equipment pieces are monitored and values are recorded to reflect the productivity and health of each machine. For example: input rate, output rate, machine temperature, speed of operation, amount of current being drawn, ...etc. This continuous data gathering results in a wealth of data, but the transformation of this data into useful knowledge remains a tremendous challenge.

A critical missing piece is information about the activity that the equipment is engaged in while the data gathering is occurring. This contextual information helps us analyse and understand the accumulated data, allowing for better equipment operation business decisions.

Our focus is on *Intelligent Activity Monitoring* (IAM), which we define as the use of an automatic solution to track the behaviour of physical equipment.[1] The word *intelligent* is selected on the basis that user-specified instructions can execute in response to what is being monitored, as a function of the current machine activity. Such responses can be useful, for example, to transmit an email message, communicate with another software module, or record information into a database in a timely fashion.

1.2 Syncrude partnership

Cooperation with Syncrude Canada Limited was achieved to enable this thesis to be applied industrially.

Syncrude, situated in northeastern Alberta, is Canada's largest single source of crude oil and the nation's second largest producer [3]. Their final product is shipped to refineries throughout Canada and the United States.

The key steps in producing Syncrude Sweet Blend (SSB) oil:

[1] Whenever we refer to the monitoring of industrial equipment we are also referring to the relevant aspects of its environment.

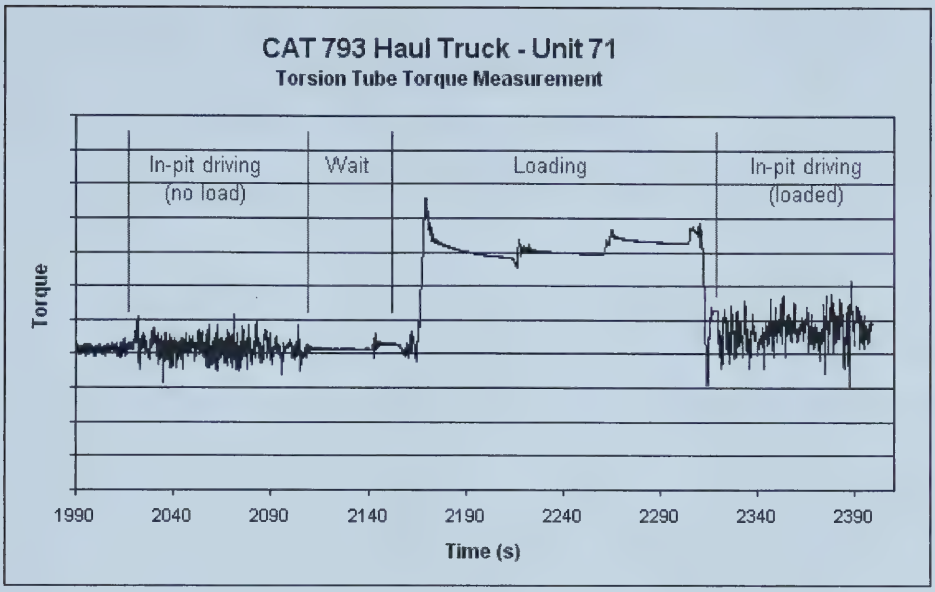


Figure 1.1: Caterpillar 793B torque measurement (used with permission, adapted from [14])

1. Mining: oil sand is dug from the ground, crushed, and hydro-transported to the extraction facilities.

2. Extraction: hot water and a light hydrocarbon treatment are used to extract the bitumen from the oil sand.

3. Upgrading: bitumen is upgraded from a tar-like oil into a straw-coloured oil that is low in sulfur.

The predominant mining strategy used to retrieve oil sand is known as 'Truck and Shovel'. This is in reference to the shovels that dig the oil sand, and the trucks that carry it to the crushers.

This work focuses exclusively on applications in mining. We say no more about extraction and upgrading.

1.3 Motivation

In Figure 1.1 we have included a graph from a Syncrude progress report. The plot identifies four separate activities[2] for understanding the torque dynamics of a Caterpillar 793 haul truck experiment. The four activities are: *In-pit driving (no load)*, *Wait*, *Loading*, and *In-pit driving (loaded)*. The activities were manually recorded during the course of the study. Without this breakdown of activity, there would be no way to explain the fluctuations in the measurement results.

[2]We refer to 'state' and 'activity' as interchangeable words.

From another perspective, contextual recordings can be used to post-analyse a process on an activity-by-activity basis. Consider questions of the form:

- What is the average amount of fuel consumed per activity?
- What is the average number of warning alarms per activity? Which activity generates the most warnings?
- Which activity causes the most deterioration (of some measurable component)?

Answers to the above can be instrumental in showing where future research efforts should be concentrated. An activity in question can potentially be further decomposed with a more refined monitoring solution, and the process repeats itself.

The idea for *Intelligent Activity Monitoring* sprung from the observation that researchers in the field were manually recording state information during the experiments they conducted. The question became: can this same state information be recorded automatically? An automated solution offers the following advantages:

- Higher Precision. Event precision can be reported to what is *measurable*, whereas manual recordings are limited to what is *observable*. The precision is more specific in two regards:
  1. Time-stamping can be as accurate as the computer clock. With manual recordings the accuracy is often limited to the nearest minute, as it is cumbersome to record to the nearest second.
  2. Sensors provide quantitative values. Knowledge in the form "turning 12 degrees to the right" is generally more useful than "slight turn to the right," as a human might report it
- Increased Consistency. A computer is consistently objective, offering a non-biased view on what constitutes a given event. This ensures that an experiment can be repeated at a later date, with the results being directly comparable. With humans, bias may occur if the observer becomes tired near the end of a long shift, or changes disposition between project days, or if alternate observers are used to record the activities. Different researchers, perhaps working on separate projects, may have subjectively different interpretations on when the same event has occurred. For examples, what constitutes a left turn?
- More Variables. A computerized tracking system increases the number of variables that can be monitored *simultaneously*. The physical process of making records imposes a limit on how much data can be observed at once.
- Longer Experiments. Once a software system has been correctly set up for a given *operation*, it can potentially be run for consecutive weeks, months, or years.

As well, the above listed considerations are necessary for reliably incorporating contextual data into an optimizing control system. For example, consider a mining truck that stiffens its suspension while loading and relaxes its suspension while driving. Such an active suspension system requires an accurate understanding of what the machine is doing at any time.

1.4 Purpose

Intelligent Activity Monitoring can be usefully employed for the following purposes:

- Classifying contextual information. This is the purpose if the program is used to automate the recording of contextual information for engineering experiments. As a second example, contextual information is the primary goal if the program is used for understanding the remote environment in a teleoperation scenario.

- Recording the amount of time spent in each activity. This is the situation in cases where the tool is employed to learn how frequently (or what percentage of the time) particular activities occur in practice. The immediate results of the tool are not necessarily used; only the resulting statistical breakdown is significant. With this usage, the IAM software serves as a yardstick. It can measure efficiency of some *operation* today, and can then be used to re-measure the efficiency whenever an upgrade has been performed.

- Triggering responses. In this usage, the role of the monitoring application is to detect and respond to particular events. Most frequently the response is merely to trigger an alarm, but it is capable of executing arbitrary commands.

1.5 Goal

We propose a model-based approach to *Intelligent Activity Monitoring*. Our hypothesis is that the strategies used to model software design can be reused to model industrial activities. The relevant aspects of the equipment/process of interest are captured within an *Activity Model*. Each *Activity Model* consists of a finite number of states and a set of transitions between them.

Our goal is to create a generic computing tool that allows us to execute *Activity Model*s. Anything that can be sensed or processed by computers can be inputted into the software. The IAM tool interprets the input and, as directed by the *Activity Model*, outputs the current activity being carried out along with a timestamp. The tool should also be capable of triggering user-definable responses to particular events.

For concreteness this thesis will focus on oil sand mining applications, although we believe the approach to be sufficiently general as to be useful in other domains as well.

1.6 Overview

This thesis is organized as follows. Chapter 2 reviews related work. Chapter 3 presents our tool from a user's perspective, explaining the modelling language proposed and how the models map into program syntax. Chapter 4 is a demonstration of our tool, applied to Syncrude Crushers. Chapter 5 presents our tool from a developer's perspective, revealing the program structure and discussing what pitfalls were encountered along the way. Chapter 6 concludes the work with a summary.

Chapter 2

Related Work

Our approach uses a subset of the Statecharts modelling language for the monitoring of industrial activities. There is very little in the literature that addresses this combination. This chapter examines other similar work with respect to monitoring and modelling, to offer credibility to our model-based approach and applicability of the resulting tool.

2.1 Monitoring systems

There is a growing interest among a variety of fields for implementations of monitoring systems. This section presents a sample of recent articles related to activity monitoring.

A commercialized ambulatory activity monitoring tool is investigated by Aminian et al. [4]. They evaluate the use of the 'Physilog' tool for recording the activity of human patients equipped with accelerometry sensors on their thigh and chest. Their results indicate that the tool can effectively discriminate between sitting, standing, lying, and locomotion activities. Further, they point out a significant discrepancy between Physilog's recordings and patient self-assessments. Although Aminian et al. report on human activity, and we are concentrating on equipment/process activity, there is a common purpose between our work: accurate reports on activity are required.

There is a branch of research focusing on tracking objects within camera imagery. Grimson et al. [18] provide an engaging article along this theme that uses a network of cameras to detect and classify activities in a site. A "forest" of cameras observe the scene from different viewpoints, and communicate amongst each other to create a single reference point of view. They were able to identify cars, trucks, and people based on the aspect ratio of the tracked object. Further, they track the routes followed by the objects, with the argument that unusual activity can be detected based on a deviation from the normal activity for the corresponding time of day.

Fawcett and Provost [13] define *Activity Monitoring* as a problem class that monitors the behaviour of a large population of entities for interesting events requiring action. Cited examples include monitoring the behaviour of cellular phone users to detect fraud, monitoring computer user behaviour to detect intruders, monitoring news stories for investment opportunities, and monitoring

network performance. They categorize activity monitoring as being either *profiling*, meaning that activity is observed for the purpose of building a model, or as being *discriminating*, meaning that the program alarms on unusual activity.

The above work motivates the need for tools that perform *Activity Monitoring*. Our approach does not make use of image analysis, so this topic is not discussed further. Similar to the Physilog device, we use a finite number of states for modelling a situation. Like the work of Fawcett and Provost, our program monitors arbitrary streams of data for particular behaviour changes; our work is different due to our focus on constructing a clear model representation and our application towards mining.

2.2 Visually modelling industrial environments

Visual modelling formalisms specify behaviour with precision and are understandable by people of any background. Consequently they are an attractive medium for describing system behaviour.

This section briefly discusses two different visual models: Petri Nets and Statecharts. Both modelling techniques have been built from solid foundations, and both have received attention in academic as well as industrial domains. We have opted to use a Statecharts-based approach, though we credit Petri Nets as being an established alternative.

2.2.1 Petri Nets

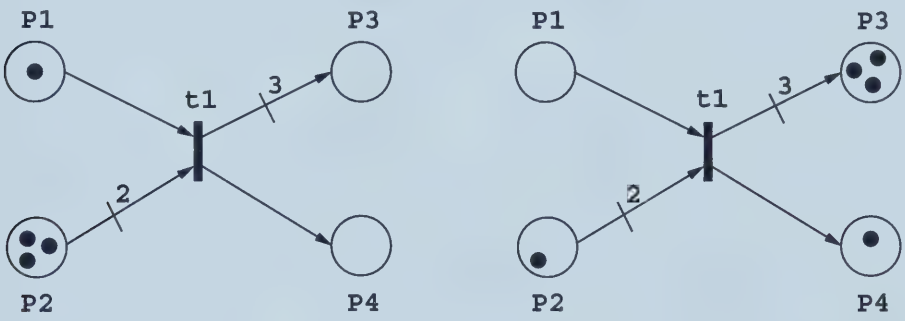
Petri Nets are a mathematical model originally introduced in Carl Adam Petri's dissertation [26]. An extensive survey has been carried out by Murata [24], which provides a solid introduction and review. Since their conception, Petri Nets have been extended with 'colour' for identifying object attributes, 'time' for describing temporal behaviour, and 'hierarchy' for composing high-level solutions out of lower-level modules [30]. Further, 'inhibitor' arcs have been suggested for increasing modelling flexibility. Petri Nets have been applied to a broad range of domains, including [24]: communication protocols, flexible manufacturing/industrial control systems, programmable logic and VLSI arrays, compiler and operating systems, and office-information systems. As such, Petri Nets have proven to be a general-purpose medium for specification.

2.2.1.1 Overview

A Petri Net is a graph with two types of nodes: *places* and *transitions*. Places are drawn as circles and transitions are drawn as bars. For discrete event models, transitions represent event occurrences and places represent the conditions necessary for the events to take place. Directed arcs travel from places to transitions, and from transitions to places. A place that has an outgoing arc pointing to a transition is called an *input place* for that transition; a place that is pointed to by a transition's outgoing arc is termed an *output place*. Input places are the preconditions of an event, output

places are the postconditions. Arcs can optionally be labelled with a *weight.* An arc with weight k is equivalent to k unlabelled arcs. Unlabelled arcs have unity weight.

When a condition is satisfied, a black dot is drawn in the corresponding place. Each black dot is referred to as a *token.* A transition t is said to be *enabled* if the number of tokens in each input place P is at least the weight of the arc between P and t. An enabled transition *t1* may or may not fire, depending on whether or not the corresponding event takes place. The model decides when events occur, according to the interpretation of what is being modelled. When an event occurs and its corresponding transition is enabled, the event is said to *fire.* A firing involves removing tokens from the input places and adding tokens to the output places. The weight of the connecting arcs specify how many tokens are to be removed/added. To be precise, on the firing of transition *t1*, each input place P1 connected to transition *t1* by an arc of weight *w1* will have *w1* tokens removed; each output place P2 connected to transition *t1* by an arc of weight *w2* will have *w2* tokens added. Figure 2.1 demonstrates a Petri Net before and after the firing of transition *t1.*



(a) Before firing *t1*

(b) After firing *t1*

Figure 2.1: Firing a Petri Net transition

A *marking* of a Petri Net gives the currents state of the modelled system. A marking specifies the number of tokens within each place. A common representation is to use a vector with m components, where m is the number of places. The number of tokens in place P corresponds to *M(p).* In Figure 2.1, part A has marking $[1,3,0,0]^T$ and part B has marking $[0,1,3,1]^T$. A Petri Net's initial marking is denoted M_0.

A transition without input places is called a *source transition.* Source transitions are unconditionally enabled. A transition without output places is called a *sink transition.* Sink transitions consume tokens (when the connecting transitions fire) but do not produce any.

Inhibitor arcs. An *inhibitor arc* is a special type of arc that tests whether an input place has *less than* a certain number of tokens. Graphically, the receiving end of an inhibitor arc is a small circle instead of an arrow head. An inhibitor arc prevents a transition from being enabled (and

hence firing) whenever its input place contains too many tokens. A transition that is connected to by one or more inhibitor arcs is permitted to fire if and only if

a. All of the normal (non-inhibitor) input places have sufficient tokens, and

b. The number of tokens in the inhibitor arc connected places is *less than* the weight of the inhibitor arc. Unlabelled inhibitor arcs have weight 1.

The firing of a transition does not affect the number of tokens in the inhibitor arc connected places.

Figure 2.2 is a modification of the graph in Figure 2.1. Input place P2 is now connected to transition *t1* via an inhibitor arc. The marking in the figure does not have any tokens in places P1 or P2. Transition *t1* will therefore be enabled if P1 acquires a token first, and will be disabled once P2 acquires a token.

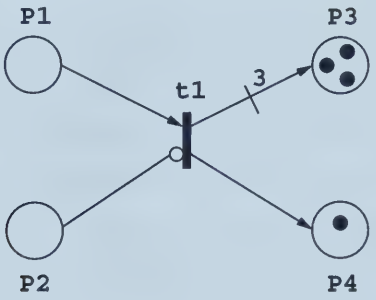


Figure 2.2: Petri Net with an inhibitor arc

Colour. A *coloured* Petri Net is one that associates attributes to tokens. Tokens are usually representative of objects, whether it be people, equipment, or product. For example, if a truck is modelled by a token in the Petri net then we may want to represent the capacity, registration number, location, etc. of the truck [30]. The colour is a metaphor; it is usually illustrated in a manner similar to Figure 2.3. Complicated models may involve objects of different types, and coloured Petri Nets are designed for this reason.

The attributes of a coloured token may change as it moves throughout the system. A coloured Petri Net requires transitions to map input tokens to output tokens. Transitions may be setup to only fire on certain input token colours.

Timed Petri Nets. Time constraints can be added to Petri Nets in various ways. Commonly tokens are time-stamped to indicate when they are available for processing by the next place/transition. Each place and/or transition can be associated with a delay. Token delays can be described by a fixed value, an interval, or a probability distribution [30].

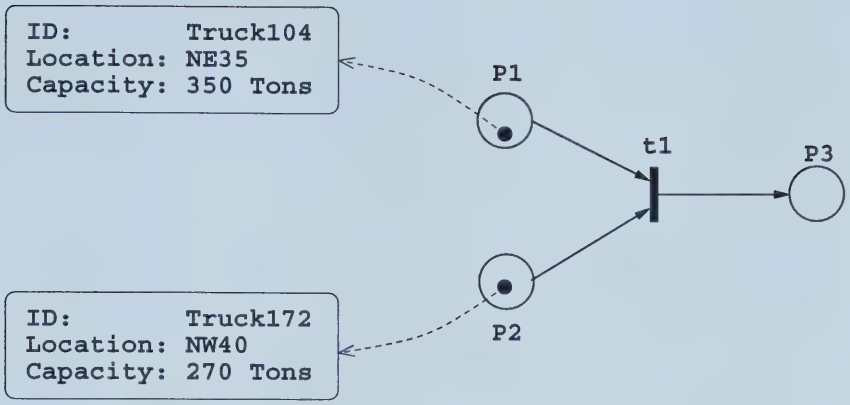


Figure 2.3: A coloured Petri Net

Hierarchy. A *hierarchical* design works off the assumption that sub-systems can be re-used, either again in the same application or in a different one. To introduce hierarchy all that is necessary is an understanding that a certain arrangement of states and transitions constitutes a *module*. Then this module can be used multiple times by referring only to its name. When necessary, each use of the module can be replaced by the full details of the module's definition.

An example is provided in Figure 2.4. Figure 2.4(a) contains one of the commonly-used modules put forward by Zhou and Venkatesh [34]; the module represents a fault-prone resource. We will refer to this module as the *faultModule*. Place P4 represents the availability of a resource, place P5 represents the resource being repaired. Transition *t3* fires when the resource breaks, transition *t4* fires when the resource has been fixed. Figure 2.4(b) shows a sample Petri Net that incorporates two instances of *faultModule*. Figure 2.4(c) is the same as part b with the full details of *faultModule* being displayed.

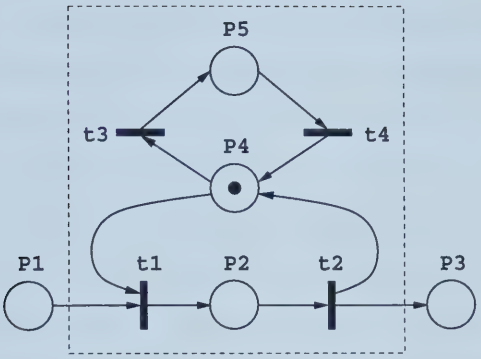
Various other expansions to Petri Nets have been employed, but their discussion is beyond the scope of this work.

2.2.1.2 Modelling examples

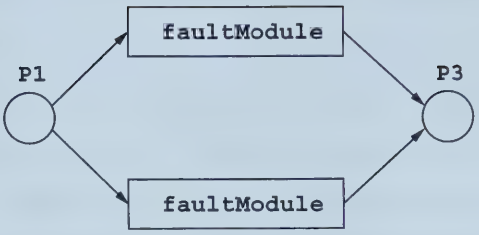
This section includes but a couple of the case studies where Petri Nets have been applied to represent industrial systems.

Interval timed coloured Petri Nets are used by van der Aalst and Odijk [31] to simulate and analyze the Dutch railroads. Each section of railroad track is associated with a bounded delay interval to account for variability in activity durations. They focus on (1) throughput and waiting times of trains in railway stations and (2) the occupation rates of sections of track. After analyzing the model using the (MTSRT) method described therein, they were able to ascertain upper and lower performance bounds.

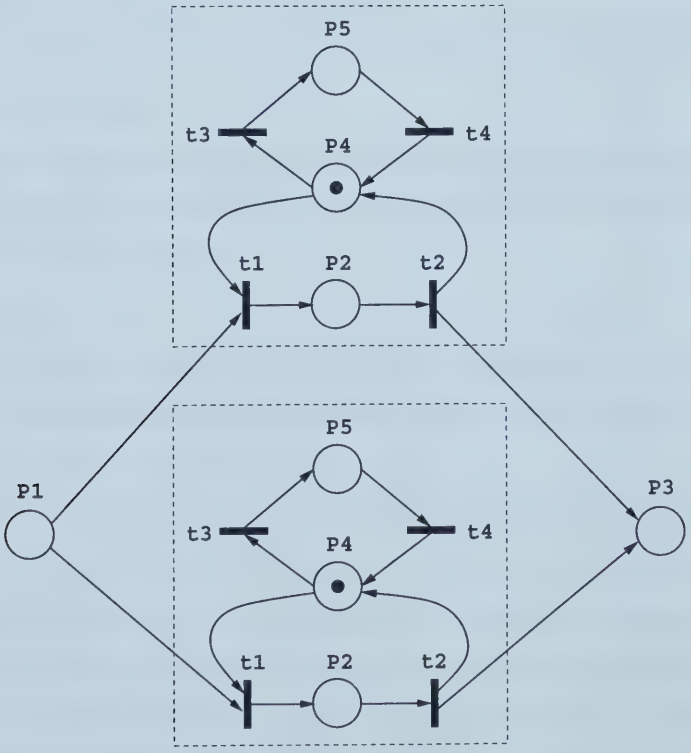
Bulitko and Wilkins [7] propose the use of Extended Petri Nets (EPNs) within a Blackboard



(a) Definition of *faultModule*



(b) Petri Net with two instances of *faultModule*



(c) Petri Net showing full details of *faultModule* instances

Figure 2.4: Hierarchical Petri Net

framework. A Blackboard framework typically consists of three steps: a deliberation step where actions are proposed, a scheduling step where proposed actions are evaluated, and an execution step where the selected action is carried out. The paper suggests using EPNs for the scheduling step.

To demonstrate, they model fire and flood related phenomena that take place aboard a DDG-51 class destroyer. They apply their EPN framework as part of the Minerva-5 expert system, used to train and direct a ship's crew through a crisis. The EPN model is consulted to learn what possible next states may result from a particular action being followed, given the current status of the ship. These next states are then evaluated by a separate module, dubbed the Static State Evaluator, and the highest-scoring action is selected. To evaluate the performance of their scheduler, scenarios were run within the DC-Train ship damage control simulator used at the Surface Warfare Officer School (SWOS) in Newport, R.I. Out of 160 ships, Minerva-5 lost 21 ships while its predecessor Minerva-4 lost 28. Minerva-5's scheduler represents context with EPN models; Minerva-4 uses a context-free rule-based scheduler. The implication is that contextual data can be valuable as part of a decision-making process.

Other industrial uses of Petri Nets can be found in Zhou and Venkatesh's book [34].

2.2.2 Statecharts

Statecharts, an expansion on finite state machines, were initially proposed through the original work of Harel [19]. Statecharts make two major advancements: they allow sub-states within states, and they add parallelism to state diagrams.

2.2.2.1 Overview

Statecharts define AND-states and XOR-states. An XOR-state allows exactly one of its sub-states to execute. An AND-state allows multiple OR-states to execute. An XOR-state introduces sub-state decomposition; AND-states introduce parallelism.

Transition arrows between states are effected by Event-Condition-Action rules (ECA rules). ECA rules are expressed according to format: `E[C]/A`. The translation is that the transition will fire on the occurrence of event E if the expression given by condition `C` evaluates to true at the time. When a transition fires, control transfers from the current state to the state pointed to by the transition arrow. Also, as part of the transition firing, the action `A` of the associated ECA rule executes. Any of the three components of an ECA rule can be omitted. Omitting the event E means the transition is always enabled; omitting the condition `C` means that the transition fires whenever the event E triggers; omitting the action `A` means that there are no code segments linked to the firing of the transition arrow.

The remaining aspects of Statecharts will be introduced through the example shown in Figure 2.5. State `A` is an AND-state, consisting of two XOR-states called `B` and `C`. Default arrows, arrows whose origin is a black circle, indicate the starting state of each XOR-state. In Figure 2.5, the default

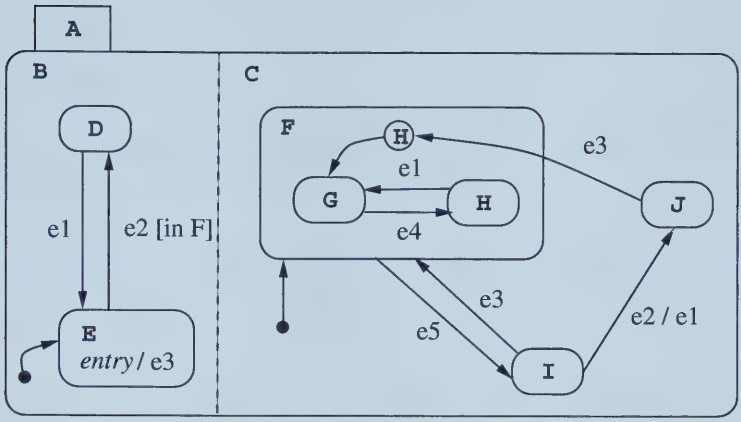


Figure 2.5: Statecharts Example

starting states of A are E and F.

State F is an XOR state with a *history connector*. The history connector is drawn as a circled H. When there is no history (i.e. the state is entered for the first time), the arrow leaving the history connector indicates the default starting state. Otherwise, if the state is entered via the history connector the sub-state that was active most recently will become active again. In Figure 2.5 a transition from state J to F will activate whichever of G or H was active at the time of the last exit from state F.

A transition exiting a parent state is given precedence over a transition exiting one of its sub-states. For example, assume that state G is active in Figure 2.5 and that events *e4* and *e5* occur at the same moment. The transition labelled *e5* would execute and the *e4* event would be ignored.

Broadcast events are events generated by one transition that may in turn cause another transition to execute. In Figure 2.5 the transition from state I to J broadcasts an *e1* event, which will trigger the transition from state D to E if state D happens to be active at the time.

Statecharts support *static reactions* that can be executed while a state is active, without causing an exit from the state. Some authors, for example Rumbaugh et al. [27], display static reactions within the state diagram. Static reactions use the same ECA format that is used for transition arrows. In addition to the usual event syntax, static reactions can be triggered by pre-defined events: *entry* and *exit*. In Figure 2.5, the *entry* event of state E specifies that the *e3* broadcast event will be generated every time state E is entered.

The transition from state E to D is coupled with the condition 'in F'. This condition evaluates to true when state F is concurrently active, and false otherwise. Similarly, events can be defined to occur when a parallel state is entered or exited.

The portions of Statecharts that are applicable to our model will be covered in detail in Section 3.2. Our approach has a number of simplifications from that described by Harel [19, 21]. The

obvious differences include:

- No history mechanism.
- No actions are associated with transitions. Instead, we link actions to the Entry/Exit code of the involved states.
- No broadcast events. Their use is not essential. For example, instead of having a state wait for a broadcast event generated by a transition, the state can wait for the destination of that transition to be entered.
- Simpler transitions. Statecharts allow transitions to be specified with multiple events and also without any events. We require each transition to have exactly one event.
- Simpler events/guards. A finite number of event types are supported (see Section 3.4), and a restricted guard syntax is used (see Section 3.5).
- Different timing semantics. This topic is discussed further below.

Statecharts are principally used for software development, to model the internal behaviour of program modules. *Activity Monitoring* is a more focused topic, and it is our opinion that not all of the mechanisms of Statecharts should be required. Our hope is that the reduced syntax will be easier to learn and use.

2.2.2.2 Modelling examples

Statecharts have received wide acceptance within the reactive systems community. Their popularity is causing their use to expand into other domains.

Wodtke et al. [33] suggest employing them towards enterprise-wide workflow management. FOLDOC [1] defines workflow as "the movement of documents around an organization for purposes including sign-off, evaluation, performing activities in a process and co-writing." Workflow management provides organization and structure to inter-office and inter-department online projects.

Leveson et al [22] develop a Requirements State Machine Language (RSML) based on Statecharts. They use RSML to specify the system requirements of an aircraft collision avoidance system called TCAS II. One of their goals was to have the resulting model be readable and reviewable by non-computer scientists; their success in this goal can be measured by the design errors that were unveiled by airframe manufacturers, airline representatives, pilots, as well as other external reviewers.

Cook and Daniels [10], in their book on object-oriented modelling, suggest using Statecharts for the development of *essential models*. An essential model is described as:

> ... an external observer of a modelled situation [within the world]. Events 'leak out' from the situation being observed and are detected by the essential model, which tracks the

changes of state of the observed situation. The essential model states which sequences of events can happen, and which cannot.

Essential models are the first of three models proprosed by [10] for software development projects.[1]

There exists similarity between essential models and the activity models that we describe in Chapter 3. Both use Statecharts to describe the dynamic behaviour of 'real-world' activities. However, the purpose of these models are very different. The intention of essential models is to better understand a problem, developing facts about the world that the (planned) software will operate within. The essential model aims to, among other things, reveal a good boundary between the proposed software and the other parts of the system. Our purpose for developing activity models is to automate them, to capture information regarding the context within which other sensor data are being collected. Activity models are *not* the base point from which more precise models are created; detail beyond what is needed to execute the models is not required.

2.2.2.3 Implementations of Statecharts

Our *Intelligent Activity Monitoring* tool is based on the Statecharts modelling specification. As such, it is appropriate to discuss other implementations of Statecharts that can be found in the literature.

i-Logix Inc. has developed a commercial version of Statecharts, dubbed STATEMATE. One of the (two) founding members of i-Logix is David Harel; consequently STATEMATE can be viewed as the official Statecharts implementation. The semantics of STATEMATE differs slightly from Statecharts, and is documented within [21, 20]. Statecharts are but one of the languages supported; STATEMATE additionally supports Activity Charts for illustrating the functional (data flow) viewpoint and Module Charts for the structural relationships. STATEMATE allows users to graphically compose Statecharts, simulate/animate them, and generate corresponding executable code.

The SMOOCHES [6] (State Machines for Object-Oriented Concurrent Hierarchical Engineering Specifications) environment supports the development and simulation of state transition models. SMOOCHES extends Statecharts by introducing exit-safe states, as shown in Figure 2.6. Exit-safe states ensure that ancestor states are "stable" before they are exited. An ancestor state is only permitted to fire outgoing transitions if all active sub-states are exit-safe. Exit-safe states are of particular interest when a physical device is being manipulated by the software, as the intermediate states may reflect intermediate motions of the device (i.e. a robotic arm). On the upside, exit-safe states allow the re-specification of sub-state definitions without modifying higher level state specifications. On the downside, they require the introduction of (a) "special events" which bypass the exit-safe status when necessary, i.e. for error events, and (b) handling methods to store pending

[1] The second and third model stages are the *specification model* and the *implementation model* for describing high-level and low-level software behaviour, respectively. At the heart of using models to guide software development is the idea that the models will lead to re-useable code and more robust software.

events when the sub-states are not exit-safe and to select among the pending events when the sub-states become exit-safe.

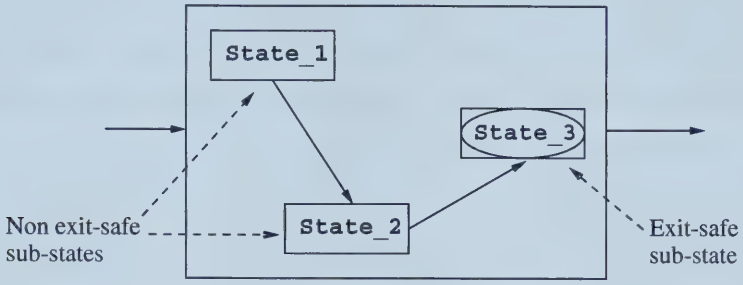


Figure 2.6: Exit-safe sub-states

SMOOCHES is coded with C++. An interesting implementation point is that the sub-states and actions can be inherited from base classes to derived classes. The analysis of a SMOOCHES model is supplemented with minimum/maximum activation durations for each state. A graphical monitoring tool allows the model behaviour to be viewed.

Our implementation does not support exit-safe states. When it is essential that ancestor transitions do not execute during select sub-states, the `IN` guard function can be used (see Section 3.5). Note that we do not allow static reactions to be interrupted during their execution (see Chapter 5); any critical section contained within the code sequence of a static reaction does not require special protection.

Paul Lucas [23] developed a CHSM (Concurrent, Hierarchical, finite State Machines) compiler to parse and create executable code for a textual language system based on Statecharts. CHSM is coded in C++, and makes use of the *lex* and *yacc* language tools.

Niere and Zündorf [25] have created a graphical CASE tool for specifying software systems in UML notation. Their work supports Statecharts representations, and has the capability to generate Java code. An integrated object browser can be used to visualize a given execution. They suggest using the tool for simulating production environments prior to making changes or reconfigurations.

Our tool is not as advanced as the ones listed above. Our goal, however, has not been to compete with industry or other CASE tools; we opted for a solution that would be simple to pickup and use. The supported features are sufficient for developing monitoring applications and demonstrating their utility.

2.2.2.4 Semantics

There has been some debate among the Statecharts community as to how timing semantics are best incorporated within Statecharts [20, 22, 32]. Here we will informally summarize the semantics used by STATEMATE and how our implementation relates. Full details of the semantics enforced by STATEMATE can found within [20, 21]. Full details of our execution model is described in Chapter 5.

STATEMATE semantics. A status is a snapshot of the system. A status consists of the values of all data-items and conditions, a listing of the events that are currently active, and a listing of all the states that are active. The listing of states within each status always makes sense; the ancestor of every active sub-state is also active, the sub-state of every inactive ancestor is also inactive.

Each time the status changes, a *step* is said to be taken. A step corresponds to the execution of all transitions and static reactions that have been enabled by the previous step. The firing of a transition modifies the disposition of the system: states are entered/exited, broadcast events are generated, condition expressions and data-items are updated. These system changes may trigger internal events in the parallel components of the Statechart, which may in turn invoke more internal events, thus triggering a *chain reaction.* The system is said to be *settled* when there are no more internal events that need to be responded to. The sum of all the steps taken between an external change and the time that the system settles is referred to as a *superstep.*

STATEMATE adopts the policy that changes in a step are not responded to until the next step. Each event, once generated, is active for exactly one step: the following step. The execution order of transitions and static reactions within a step is insignificant because the values of the data-items being acted upon are not updated until the next step.

STATEMATE defines two execution models, synchronous and asynchronous.[2] In the synchronous model the rising or falling edge of the CPU clock indicates the start of a step. External changes and timeout events are detected at the first clock tick following their occurrence. It is therefore possible for the system to be in the middle of a superstep, not yet stabilized, when a subsequent external event is detected. The asynchronous model, on the other hand, waits until the system has reached a stable state before responding to additional external events. The asynchronous model will respond immediately if it is not in the middle of a superstep.

IAM semantics. When comparing against the STATEMATE execution, our approach is quite similar to the asynchronous time model. The IAM software listens for external events to occur. Multiple events can occur at the same time. After receiving a "batch" of events from the environment, the program executes until it has settled itself into a new combination of states that is stable. It then waits for more external events, and the cycle continues.

Unlike STATEMATE, we do not assume a step-wise semantics. Changes to data-items are immediate. Consequently, the order of commands within static reactions (which we refer to as 'response actions') is significant. This semantics resembles what was initially proposed by Harel [19].

There are three possible outcomes of an event occurrence. The first is that the event is responded to by one or more states. The second is that the event is not currently being waited for by any states, and therefore gets discarded. The third is that the sub-state(s) waiting for the event become inactive before given the opportunity to respond to the event, in which case the event is discarded.

[2]Here we are only considering STATEMATE's dynamic test tool, ignoring the simulation tool.

This latter case can occur if the source state(s) are waiting for more than one event at the same time, or if the parent state transitions before the sub-states are given a chance to.

2.3 Discussion

2.3.1 Modelling language

More often than not Petri Nets have been used for industrial modelling. State diagrams have traditionally been dismissed on the grounds of an inevitable state space explosion. One alternative solution is Statecharts; they provide a method of diagramming with states that avoids the state explosion problem. This thesis investigates the use of Statecharts for monitoring applications.

2.3.2 Purpose

The following are common industrial uses for models:

Simulation. These models generally answer "what if" questions. They often help decide how new equipment is to be installed and used before physical changes are made. Simulation models can also re-create situations after the fact, to help assess why something went wrong. A common attribute of simulation is that time is advanced by the software whenever doing so does not affect the outcome. Examples of simulation modelling include [31] and [7].

Planning and Scheduling. These models help to show what arrangements lead to optimal (or near-optimal) production. For example, how many mining trucks and shovels should be operating at once? What routes should each truck travel?

Monitoring. These models are executed to describe some aspect of the world in real-time. They report on the current status of the plant, possibly evaluating the performance of some person/equipment. This approach is in line with the work of [4] and [13].

Control. These models dictate how the machinery is to respond to the changes in its environment. Control applications often contain a monitoring sub-component.

Software Design. These models design how a complex software system is to function. These software designs are often intended for review by people other than developers alone. Where necessary, i.e. for safety critical applications like TCAS II, these designs may be subjected to formal analysis procedures to ensure that all specifications and constraints have been met. Examples of software design include the contributions on workflow management [33], software development [10], and aircraft collision avoidance [22].

These headings are not mutually exclusive. The work on aircraft collision avoidance [22], for example, might be classified as either of *Monitoring* or *Software Design*. The 'monitoring' title reflects the manner that the state of the plane and nearby planes is kept up-to-date. A 'software

design' classification is also appropriate when you consider that software executes according to the model's specifications, and that the model ties together with existing systems. Regardless, the above points show many of the different reasons for why a model is constructed in the first place.

Intelligent Activity Monitoring is intended to help with the monitoring role, and also for post-analysis simulations where monitoring data has been collected. Our "designs" are limited to describing how an operation is to be monitored: what states to use, when to transition between states, what state sequences are possible. IAM diagrams can be used to communicate among professionals of different expertise the manner that the system state is derived.

Our support for actuation is in the form of user-defined *response actions* (Section 3.6). Response actions are intended to make the monitoring tool more effective, i.e. by recording to a database or triggering an alarm. In contrast, control responses would directly affect the operation being observed.

2.3.3 Modelling the world

Cook and Daniels [10] make the point that modelling the world and modelling software are fundamentally different. They stress that the world is unpredictable, as the future cannot be absolutely determined from the past. In comparison, software is predictable because its behaviour is determined by its own code. Cook and Daniels refer to the world as being *partially* predictable because aspects of it can be described with factual statements. For example: once the sun has risen, it will not rise again until after it has set.

Our objective is to monitor cycles in industrial behaviour, using either live or stored data. With stored data, the thresholds that cause transitions in the model can be based on an analysis of the data being used. In the live case, the anticipated behaviour of each cycle is estimated after examining some history of the equipment in question. History data can be viewed as another example of *partial* predictability - the future does not guarantee that surprises will not occur.

Automated monitoring of the real world consists of (a) sensors being in place that are configured correctly and (b) software that rationalizes the current state given sensor input. The former item deals with signal analysis. Signal analysis tends to weigh heavily on the application of known algorithms. Visual models are more central to the latter item, as they show how and when state changes occur given different event sequences. Visual models become impractical when they need to additionally explain how each sensor value is analyzed. Particularly so when there are a large number of sensors or when dealing with sensors that have noisy data.

Our architecture cleanly separates signal analysis from state processing. Signal analysis can either be deferred to external software or implemented as supplementary Knowledge Source (KS) modules. KS modules are introduced in Section 3.3.2. Each KS module executes with a separate thread of control and feeds input into the *Activity Model* software. The *Activity Model* software, in turn, derives the overall system state.

Chapter 3

Activity Programs*

Each *Activity Model* is a blueprint that defines how industrial equipment is to be monitored. According to Merriam-Webster, an activity is "an organizational unit for performing a specific function" [2]. *Activity Model*s represent each activity within a machine's work cycle as a state; transfer through the work cycle is represented by transitions between the states.

An *Activity Program* is an implementation of a corresponding *Activity Model.* An *Activity Program* is executable when used in conjunction with our IAM software package.

The basic vocabulary of *Activity Models* is defined in Section 3.1. The visual formalisms of *Activity Models* are introduced in Section 3.2. Section 3.3 illustrates how an *Activity Program* fits into the overall structure of our tool. Sections 3.4, 3.5, and 3.6 continue by describing the *Activity Program* syntax of supported events, guards, and response actions. Section 3.7 shows how to translate an *Activity Model* into an *Activity Program.* Section 3.8 demonstrates the incremental construction of a solution model. The chapter finishes with a brief summary.

3.1 Definitions and terminology

We now define the fundamental concepts used in *Activity Models*: states, events, transitions, and guards. The definitions presented herein are similar to [10, 21, 27] but we repeat them for the sake of completeness.

3.1.1 States

A state is the disposition of a system. Whenever a particular state is active, the system has a consistent behaviour for responding to events. Each state has duration; a state corresponds to the interval of time between two events.

Examples of states can include "shovel bucket full," "truck driving in reverse" or "brake pedal depressed."

*Portions of this chapter come from my unpublished Syncrude progress report [15].

3.1.2 Events

Events cause the model to switch from one state to another. An event is defined as:

- An indication that something has happened
- An instantaneous signal; one with no duration.

As noted by Rumbaugh et al. [27], "nothing is really instantaneous; an event is simply an occurrence that is fast compared to the granularity of the time scale of a given abstraction." Further, [10] comments:

> [events] have not yet happened or they have already happened; they can never be happening. We can know that an event has occurred only by detecting its effect on our model. ...

Examples of events include "shovel has made contact with the ground," "ignition key turned," "email acknowledgement received from client" or "sensor input available."

3.1.3 Transitions

Transitions represent the decision-making of the model. A transition specifies when it is time to leave one state and enter another. Each transition is associated with an event; a transition is triggered on the occurrence of the event. When a transition is triggered, and there is no guard (see below) to prevent its execution, the transition is said to *fire* and the model changes state.

Often the transition events are external in nature, and asynchronous as far as the activity model is concerned; the timing of the events cannot be predicted. This is the case, for example, with changes in sensor values. However, some events, most notably timeout events, are internal since their occurrence can be predicted.

3.1.4 Guards

A guard[1] is a Boolean function that enables/disables transitions. A transition triggered by an event will be fired if and only if the guard expression evaluates to true at the time of the event. A guard reports on the status of some part of the system. For example, "engine temperature is above 20 degrees" or "the truck's box is raised."

As observed in [10], guards can be used for two purposes:

1. To select one of many transitions leaving a state, where the transitions are waiting for the same event and are each associated with distinct guards. It is a design error if the guards permit more than one transition be to fired by a given event.

[1] In [19] and [21] *guards* are referred to as *conditions*. [10] put forth the term *guard*, which is more appropriate with our usage.

2. To indicate pre-conditions. In this case, the guard(s) indicate the situation(s) in which the event can meaningfully occur.

3.2 Activity Model diagrams

Our use of a graphical specification is to design *how* an application is to be monitored, for review between peers.

To arrive at our graphical specification, we started by reviewing popular software design methodologies. Statecharts, originally proposed by David Harel in [19], illustrate the behaviour of a system and seemed the most suitable for our objectives. The diagramming techniques that we have adopted are a subset of the techniques offered by Statecharts. Our goal was to capture the aspects of Statecharts that were essential for building *Activity Models* without making the resulting language more complex than necessary.

3.2.1 Basic states

A basic state is a state that does not have any sub-states [21]. Basic states are shown as rounded boxes with an identifying label. See Figure 3.1 for an example.

Figure 3.1: A state without sub-states

3.2.2 State transitions

Transitions between states are drawn with a labelled unidirectional arrow (Figure 3.2). The label can consist of the name of an event and optionally the name of a guard. If no guard is used, the transition immediately fires on the occurrence of the event.

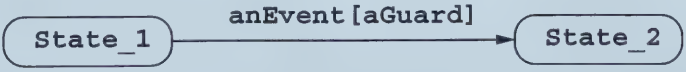


Figure 3.2: Format of a transition between states

A given state can have multiple transitions defined. Each transition can only depend on a single event. Eligible event types are listed in Section 3.4. The format of guards is detailed in Section 3.5.

3.2.3 Cluster states

A cluster[2] is a state with sub-states nested inside it (Figure 3.3). While a cluster state is active, exactly one of its sub-states is active. A sub-state can itself be another cluster state.

Whenever a cluster is entered, its starting sub-state must be specified. This can be done either by:

A. Having the incoming transition arrow point directly to a sub-state, or

B. Having the incoming transition arrow point to the cluster state, and using a default transition arrow to indicate the starting sub-state

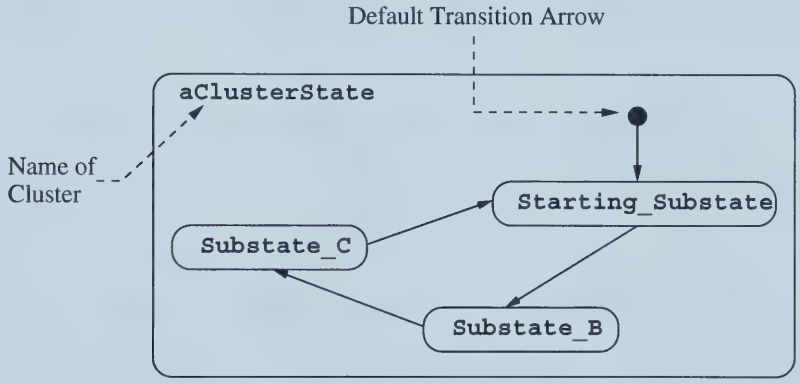


Figure 3.3: The format of a cluster

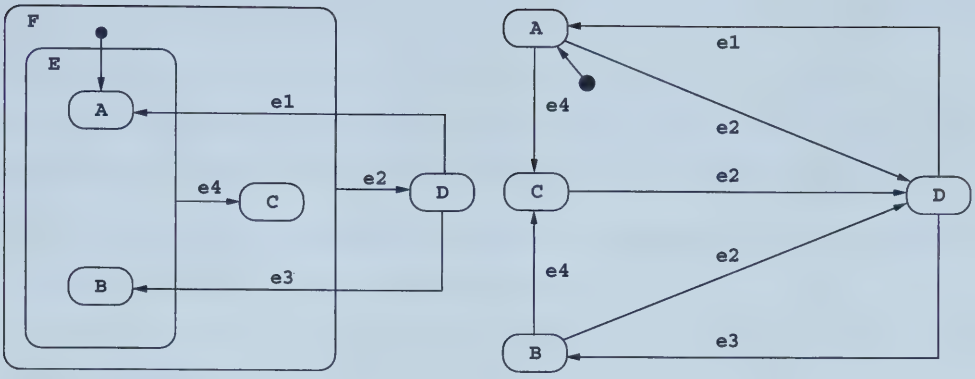
Each cluster state defines exactly one default transition arrow. A default transition usually leads to a sub-state in the first level of the state decomposition, but it can be made to directly enter a state on a lower level as well [21]. For example, in Figure 3.4 (a) the default starting state is specified to be `A`.

Clustering carries two main advantages:

1. **Reduced model size.** Transition arrows leaving a cluster apply to all of its sub-states as well. Using Figure 3.4 (a) as an example, assume that state `E` is active. Regardless of whether sub-state `A` or sub-state `B` is also active, on the occurrence of event *e4* the transition to state `C` will fire. As a result of this property, clustering often reduces the number of transitions in the diagram.

2. **Modular design.** Clusters can be defined from either a top-down or a bottom-up perspective. Top-down design proceeds by first defining the more general states, and then recursively

[2]In [21], David Harel et al. refer to clusters as *hierarchically arranged states*. Sub-states are called *descendents* or *children*; super-states are spoken of as *ancestors* or *parents*.



(a) Using cluster states (4 transition arrows)

(b) Not using cluster states (7 transition arrows)

Figure 3.4: The same diagram drawn with and without clusters

refining the model with additional sub-states. Bottom-up design clusters together states with similar behaviour to form a common parent state.

Assuming automated tool support, designs can be viewed with varying degrees of abstractness by zooming-in (showing sub-states) and zooming-out (hiding sub-states). By zooming-out, the internal workings of a cluster are not viewable, and the cluster itself can be treated as a black box.[3] Consider, for example, a technician focusing on the exact details of a particular process being monitored, while the plant manager keeps track of the higher-level activity models for all the different equipment pieces in his/her domain.

Clustering imposes a priority scheme on its outgoing transitions: priority is given to the outermost cluster.[4] For example, in Figure 3.4 (a), the transition labelled with event *e2* has higher priority than the *e4* transition.

3.2.4 Orthogonal states

If two or more pieces of equipment are to be monitored in parallel, it is tedious to merge their individual states into a model that doesn't support parallelism. For example, we might be interested in monitoring a truck's engine temperature as well as the direction of its wheels - these aspects of the truck are not necessarily directly related. Statecharts introduce parallelism through the orthogonal state, a state consisting of a number of components that execute simultaneously.

Graphically, dashed lines partition an orthogonal state into a number of components. The dashed lines can either run vertically or horizontally. Each component is similar to a cluster since

[3] By [1], a black box is defined as: *an abstraction of a device or system in which only its externally visible behaviour is considered and not its implementation or "inner workings".*

[4] This follows [19, 21]. Cook et al. [10] choose the innermost transition if there is a conflict between transitions.

it contains at least one sub-state. Orthogonal states have their names attached to the state frame in a square box.

When an orthogonal state is entered, each of its components starts to execute in parallel. A starting sub-state in each component needs to be specified either explicitly with a (possibly splitting) transition arrow or implicitly with a default transition arrow. For example, consider Figure 3.5:

- On the firing of the transition labelled *enterShell*, all components start with their defaults. In this case, sub-states `B`, `C`, and `F` all become active.
- On the firing of the transition labelled *enterSubstate*, only `Component 1` has an identified starting state. The sub-states `A`, `C`, and `F` become active.
- On the firing of the transition labelled *enterSplit*, two starting sub-states are indicated. The sub-states `B`, `C`, and `E` become active.

When an orthogonal state is exited, all of the orthogonal components simultaneously stop executing. An orthogonal state can exit by:

1. The firing of an outgoing transition arrow connected to the border of the orthogonal state (Figure 3.5, transition *exitShell*)
2. The firing of an outgoing transition arrow connected to the border of a sub-state within one of the components (Figure 3.5, transition *exitSubstate*)
3. The firing of an outgoing transition arrow that is merged from the border of more than one sub-state, where each sub-state is within a separate component (Figure 3.5, transitions *exitMergeA* and *exitMergeB*). In this case, each component involved in the merge terminates independently according to the specifications of its outgoing arrow. Once all of the components involved in the merge have terminated, then the orthogonal state is exited.

Orthogonal States offer two key advantages:

1. **Expandability**: independent aspects of the activity model can be introduced as new components, without affecting the previously defined states.
2. **Clarity**: fewer states and transition arrows are needed. Figure 3.6 demonstrates how the complexity quickly becomes unmanageable when orthogonal states are not used.

3.2.5 Special transitions

A modelling language needs to be versatile. This section introduces some shortcuts that make the model easier to draw.

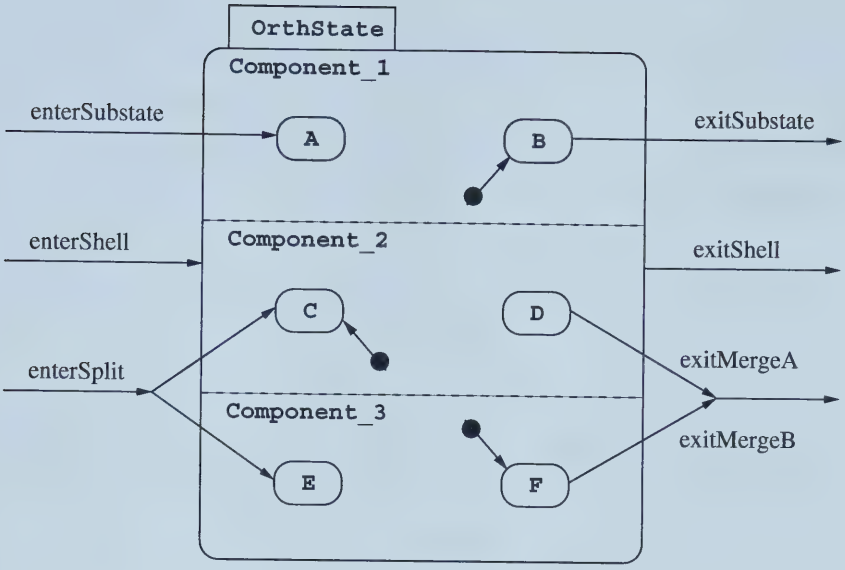
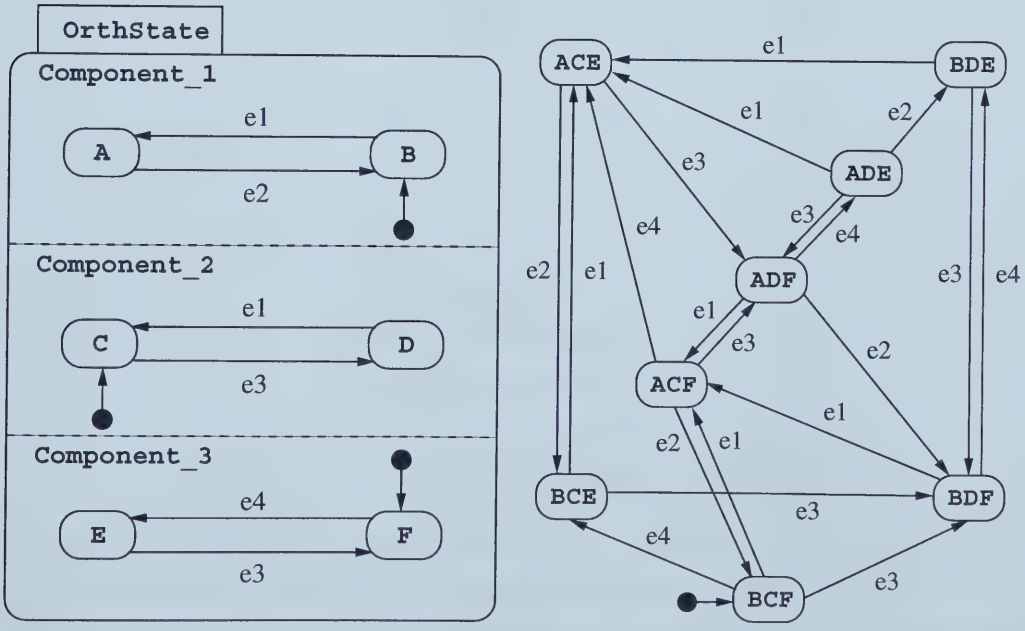


Figure 3.5: Methods of entering and exiting an orthogonal state



(a) An orthogonal state, consisting of three components (6 states, 6 transition arrows)

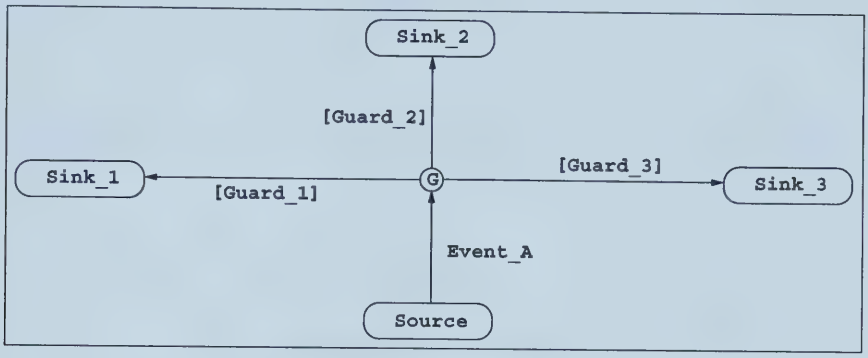
(b) Equivalent system not using orthogonal states (8 states, 20 transition arrows)

Figure 3.6: The same diagram drawn with and without orthogonal states

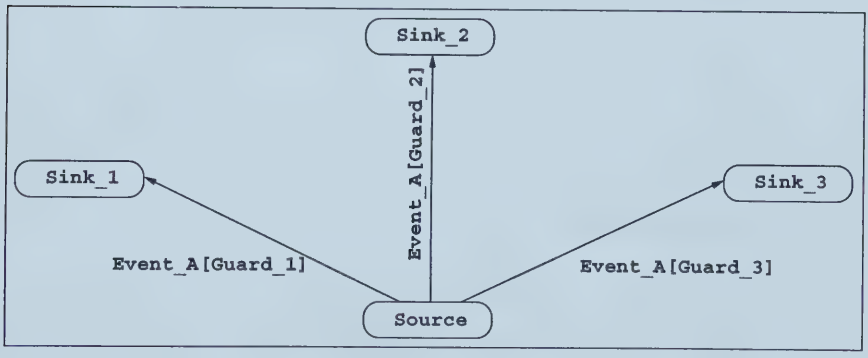
3.2.5.1 Guard branches

A *guard branch*[5] can be used when the same event triggers more than one transition, and each transition is guarded differently.

A guard branch is symbolized with a circled-G. A transition leading into a guard branch separates into multiple transitions, one for each related guard. On the occurrence of the event, the guards decide which transition (if any) will fire. Figure 3.7 shows an example.



(a) Using a 3-way guard branch



(b) Same model, without a guard branch

Figure 3.7: A comparison with and without a guard branch

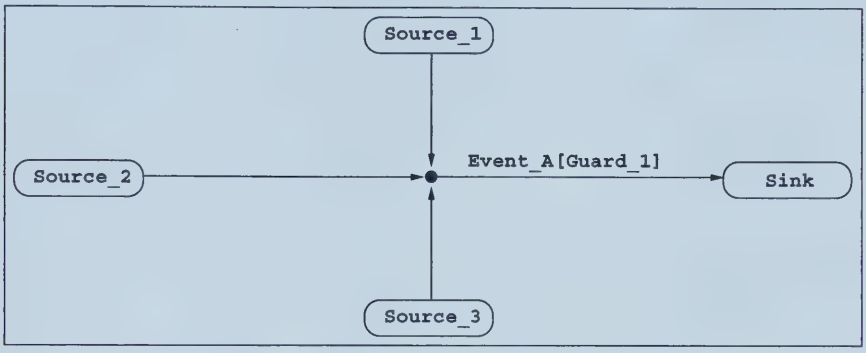
At most one guard should be able to evaluate to true. In malformed models, where multiple guards are true at the same time, the IAM software will arbitrarily pick one.

Multiple guard branches can exist along a given route. One guard branch can potentially lead into another guard branch, and etc.

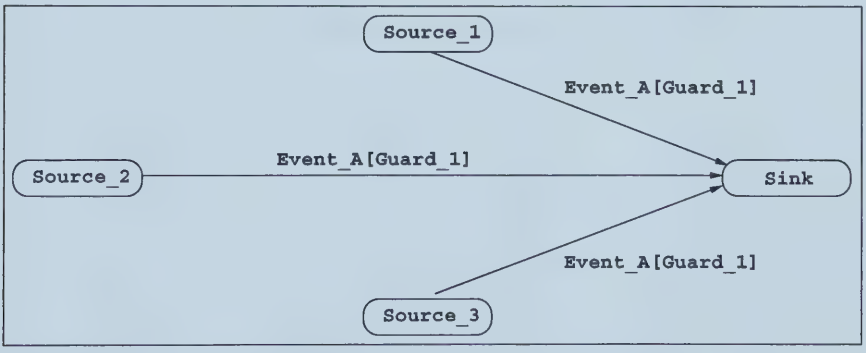
[5]Harel [21] refers to these constructs as *condition connectors*, or *C-connectors* for short.

3.2.5.2 Junction connectors

A *junction connector* brings together multiple transition arrows that trigger on the same event and share the same destination. Graphically, the connector is a black round circle that is pointed to by all of the participating transitions. In our use each connector has exactly one exit, used to specify the event and optional guard of all the transitions. Figure 3.8 illustrates an example.



(a) Using a junction connector



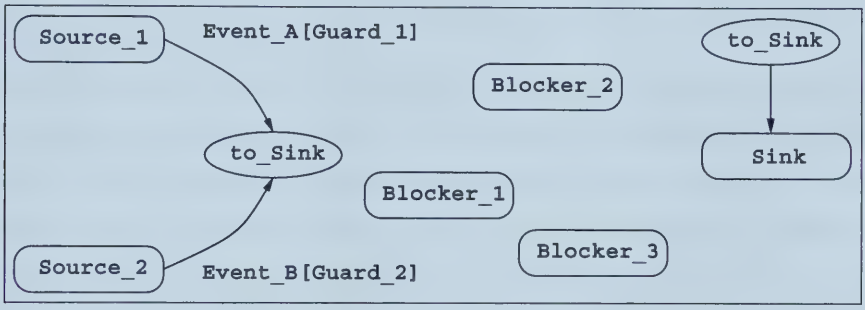
(b) Same model, without a junction connector

Figure 3.8: A comparison with and without a junction connector

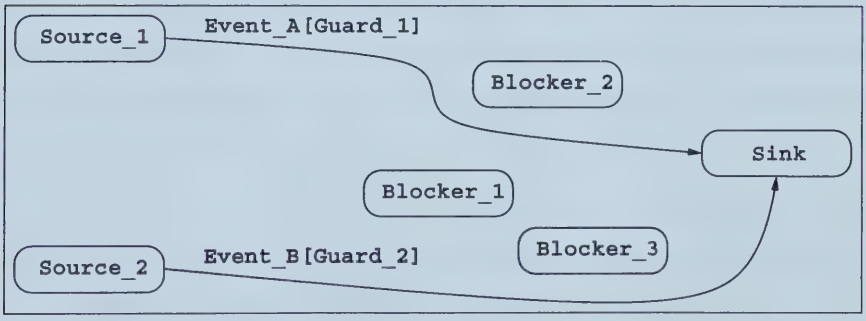
The modelling advantage of using junction connectors (and guard branches) is that it becomes obvious which events are causing entrance-to (exit-from) a particular state. Without this merging (splitting), there are more transition arrows to confuse the diagram and the input/output relationships are slightly more cryptic. As well, they save the designer from re-writing the event name multiple times.

3.2.5.3 Diagram connectors

Diagram connectors allow the transfer between two far away states, without requiring a long arrow. Diagram connectors consist of input ports and output ports. Input ports only have incoming arrows, output ports only have outgoing arrows. Both port types are drawn as an oval with a label. Ports with the same label are related. Arrows that enter an input port will exit from the corresponding output port. There is no harm in having multiple input ports for a given diagram connector, but obviously there can only be one output port. In our usage, the entire route from a source to a destination must specify only a single event. Figure 3.9 shows an example.



(a) With a diagram connector



(b) Same model, without a diagram connector

Figure 3.9: Transferring to far away states

3.3 Program structure

This section describes the framework boundaries within which we execute *Activity Programs*. The inner workings of our toolkit is presented in Chapter 5; this section is intended to provide sufficient context for a user to understand how the toolkit functions.

A high-level view of the program structure is diagrammed in Figure 3.10. The *Activity Monitoring Client* (AMC) represents the workings of our IAM software package. We start by discussing the data flow between the modules external to AMC (Section 3.3.1), and then Section 3.3.2 takes a look at the parts within.

3.3.1 The transfer from input to output

Input can come from stored files, live sensors, or over a communication link with another computer. The assumption is that the input is in a numeric form, likely representing some sensed quantity. The input does not need to be sensed directly – it can be routed through another computer that already has a sensing infrastructure in place. Further, the input does not need to be live – it can be a re-execution of previously stored data.

The *Activity Monitoring Client* tracks the current state of the industrial equipment. The computed state information is recorded to either a file or database, represented by the *Output Log.*

We did not want to complicate the monitoring application with details of each input source (i.e. timing frequency, input type, input location). The exact arrangement of input is generally application-specific. An intermediary program, the *Data Retrieval Server* (DRS), was therefore created for the task of "getting input." The DRS is the only module that needs to maintain knowledge about the input sources. The DRS is in a position where it can potentially help in filtering out noise before transmitting the data along.

The AMC is not concerned with *where* the input comes from. From its perception all input comes from a singular location: the DRS. Further, the AMC does not know *what* kind of data it is receiving – whether the input source is a real sensor or artificially created. This design allows us to modify the DRS (i.e. swap input sources) without modifying the AMC (who remains unaware of the change). This feature is convenient when moving from off-line debugging to online processing.

The *Data Retrieval Server* and the *Activity Monitoring Client* are separate programs that communicate over a socket. Consequently they are not required to be implemented in the same programming language, nor do they need to execute on the same computer.

3.3.2 The internals of a client

The AMC computes the active state(s) according to the specifications of an *Activity Program.* The mapping from a visual *Activity Model* into a textual *Activity Program* is the topic of Section 3.7. Each *Activity Program* is a script that configures a given model through a number of IAM function calls, and then turns over execution to the IAM package.

The remaining sub-systems within the AMC component are created and managed by the IAM routines. The *Input Retrieval Unit* simply receives incoming data and posts it to the *Blackboard.* The *Blackboard* is a central location for storing current sensor values. The *Blackboard* and the *Knowledge Sources* inter-operate to allow for dynamic input sources, which we refer to as "soft

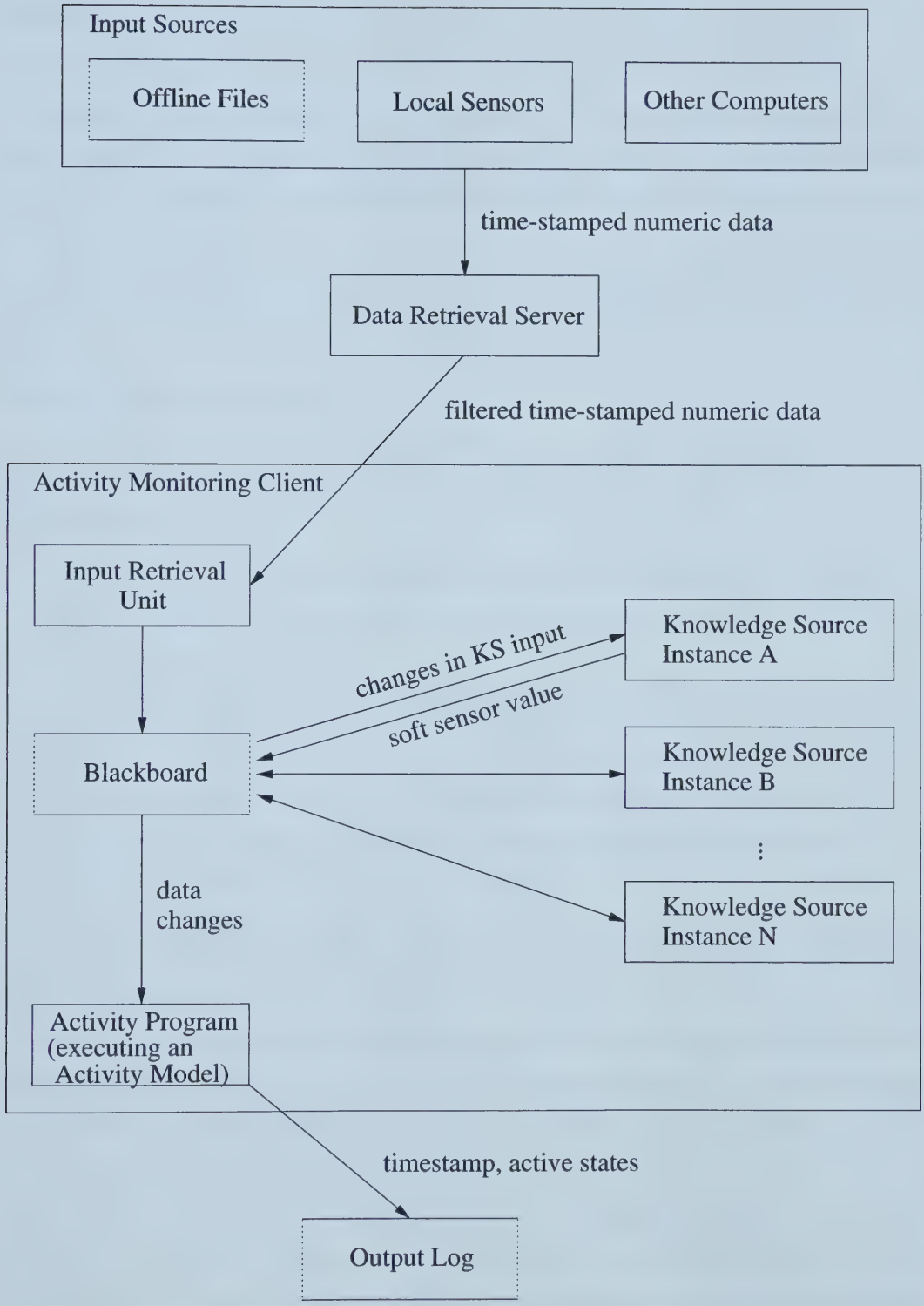


Figure 3.10: Program structure

sensors."

3.3.2.1 Soft sensors

Our definition of a *soft sensor* is any input value that cannot be sensed directly – it is derived from other sensor values. For example, acceleration is derived from velocity. It is possible for different *soft sensor*s to accept the same input(s) but calculate different output results, depending on the purpose of the *soft sensor*.

As the number of input sources viewable by the *Activity Program* increases, the tool becomes more versatile. We therefore went about finding a mechanism to support *soft sensor*s in our architecture.

3.3.2.2 Central blackboard

We use a *Blackboard*, an artificial intelligence problem-solving technique, to incorporate *soft sensor*s. Our use and understanding of *Blackboards* is based on Daniel Corkill's article [11]. In [11], the following analogy is given:

> Imagine a group of human specialists seated next to a large blackboard. The specialists are working cooperatively to solve a problem, using the blackboard as the workplace for developing the solution.
>
> Problem solving begins when the problem and initial data are written onto the blackboard. The specialists watch the blackboard, looking for an opportunity to apply their expertise to the developing solution. When a specialist finds sufficient information to make a contribution, he records the contribution on the blackboard, hopefully enabling other specialists to apply their expertise. This process of adding contributions to the blackboard continues until the problem has been solved.

In our use, the *Blackboard* is a central location for storing current sensor values. The *Blackboard* can be acted upon by multiple agents called *Knowledge Sources* (KS). Each KS, an expert by the above analogy, implements a type of *soft sensor*.

"Raw" sensor values, arriving unchanged from the *Data Retrieval Server*, are posted to the *Blackboard*. A *Knowledge Source*, upon seeing a new version of one of its input parameters, generates a new "derived" (soft) sensor value and posts it to the *Blackboard*. The *Activity Program* monitors all of the data changes that occur on the *Blackboard*, regardless of whether they are raw or derived. The *Activity Program* may respond by changing states and/or executing an action.

The above analogy indicates that contributions are made until the "problem" has been solved. In the context of monitoring, the "problem" can be considered solved once the experiment is over. *Knowledge Sources* continuously update the *Blackboard* as new sensor values are received throughout the experiment.

3.3.2.3 Knowledge sources

The purpose of a *Knowledge Source* is to transform input data into whichever format is most useful to the *Activity Program*. Two types of *Knowledge Sources* are currently supported:

1. **Trend**: Estimates the short-term trend of a data source by finding the slope of a best-fit line through the most recently collected data points. This is useful for detecting sudden increments or decrements.

2. **Discretization**: Classifies a data source according to which interval its value falls within. For example, we could assign the steering angle 0-60 as being "*turning left*", 60-120 as being "*going straight*", and 120-180 as being "*turning-right*". Then the *Activity Program* could make decisions based on {*turning-left*, *going-straight*, *turning-right*} instead of comparing against the raw sensor values directly. The goal of this KS is to keep the *Activity Program* specification at an abstract level.

A *Knowledge Source* is an object type that can be instantiated by specifying a name, a type, input data source(s), and an output data source. The same *Knowledge Source* type can be instantiated multiple times for different purposes. The output of one *Knowledge Source* can be the input to another.

The decoupling between *Knowledge Sources* allows the architecture to be easily expandable: *Knowledge Source* types can be added and removed as appropriate. A newly desired *soft sensor* (i.e. implementing a neural net) can be introduced as a new *Knowledge Source* type without affecting the existing KS modules.

3.4 Event specification

This section outlines the syntax and semantics of events as they are supported in our implementation of *Activity Programs*. The supported event types are listed in Table 3.1.

Events can be used to label transition arrows and/or response actions (see Section 3.6). Events are not fully specified directly inside the *Activity Model*. Instead each event invocation is assigned a name, and this name is referenced directly from the *Activity Model*.

It is common for multiple events, sometimes of the same type, to be pending at the same time. While the model is executing, events that can potentially cause a subsequent state transition are automatically registered each time a state is entered. If a newly entered state has ancestors, then the outgoing transitions of each ancestor are also registered. When a transition is fired, all previously registered events are cleared away.

The **Timeout** event occurs once the specified time expires. The **OrthEnter** and **OrthExit** events occur when the specified state is entered or exited, respectively. The **ThreshExceeded** event is used to detect when a *Blackboard* variable exceeds a fixed threshold. The value assigned to

Timeout (*timeString*) *timeString*: has format hh:mm:ss
OrthEnter (*stateName*) *stateName*: a state in an orthogonal component
OrthExit (*stateName*) *stateName*: a state in an orthogonal component
ThreshExceeded (*blackboardVariable, dir, thresh [, persistence]*) *blackboardVariable*: identifies one of the sensor variables on the blackboard *dir*: one of { >, >=, <, <=, ==} *thresh*: a numeric value *persistence*: an optional numeric value, the default is 1
RangeExceeded (*blackboardVariable, base, tolerance [, persistence]*) *blackboardVariable*: identifies one of the sensor variables on the blackboard *base*: a numeric value *tolerance*: a numeric value *persistence*: an optional numeric value, the default is 1
StringComparison (*blackboardVariable, type, aString [, persistence]*) *blackboardVariable*: identifies one of the sensor variables on the blackboard *type*: one of {"equalTo", "notEqualTo"} *aString*: a string of ASCII characters *persistence*: an optional numeric value, the default is 1
Change (*blackboardVariable*) *blackboardVariable*: identifies one of the sensor variables on the blackboard

Table 3.1: Supported event types

the *dir* parameter indicates whether the event occurs when *blackboardVariable* exceeds the threshold value, falls beneath it, or matches it exactly. The **RangeExceeded** event detects when *Blackboard* values are outside a given range. The allowable range for the *blackboardVariable* is: *base* $\pm$ *tolerance*. Values outside of this range cause the event to fire. The **StringComparison** event compares *Blackboard* values against strings. **StringComparison** can be useful when there are instantiations of the discretization KS (Page 33). If the *type* parameter is set to "equalTo" then the **StringComparison** event will occur when *blackboardVariable* is assigned the same value as *aString*; if *type* is set to "notEqualTo" then the event will occur when *blackboardVariable* is assigned a value different from *aString*. If any of the **ThreshExceeded**, **RangeExceeded**, **StringComparison** expressions already evaluate to true, then the corresponding event will fire immediately. The **Change** event occurs when the *blackboardVariable* is assigned a new value. If it is continuously assigned the same value then the event will only occur once, on the very first assignment.

A mechanism is needed to prevent the threshold events from false-triggering on noisy input. Ideally the data from the *Data Retrieval Server* has already been filtered from noise, but this is not always feasible. *Persistence* helps protect against false alarms [28]. *Persistence* delays the firing of a transition, requiring an event to be recognized for a number of consecutive samples before it is considered valid. A large *persistence* value increases our confidence in the event detection, with the cost of an introduced delay before the program perceives the event.

The **ThreshExceeded** and **RangeExceeded** event types both optionally make use of *persistence*. Each of these event types is associated with a count, initially zero. If the count reaches the value specified by the *persistence* parameter, the count is cleared and the event occurs. The count is updated whenever the relevant *Blackboard* variable is modified; the *persistence* mechanism does *not* poll the *Blackboard* variable at a regular time interval. On an update, the count will be incremented if the event's threshold has been surpassed. **RangeExceeded** has two thresholds that are checked; **ThreshExceeded** has one. For both event types the count will be decremented if the *Blackboard* value falls within the normal operating range.

The **StringComparison** event can also make use of *persistence*. When the *blackboardVariable* is modified, the *persistence* count is incremented if a **StringComparison** event would normally be triggered, and is decremented otherwise.

3.5 Guard specification

Guards are expressions that evaluate to either true or false. When false, they prevent the associated transitions from firing. Guards reflect properties that exist in the environment or in an orthogonal component of the model itself. Their evaluation may change over time.

Our implementation supports guards following the syntax given in Table 3.2.

The aspects of the environment that are noteworthy and measurable are stored as *Blackboard* variables. Our toolkit allows guards to compare the values of the *Blackboard* variables.

Guard	⟶	*Expr* | (*Guard*) AND (*Expr*) | (*Guard*) OR (*Expr*)
Expr	⟶	*Term* | NOT (*Term*)
Term	⟶	*Comparison* | *Function* | guardName
Comparison	⟶	*NumFactor AnyOperator NumFactor* | *StrFactor Operator1 StrFactor*
Function	⟶	IN (`stateName`) | DONE (`componentName`)
AnyOperator	⟶	*Operator1* | *Operator2*
Operator1	⟶	== | ! =
Operator2	⟶	< | > | <= | >=
NumFactor	⟶	*Blackboard* variable | numeric constant
StrFactor	⟶	*Blackboard* variable | string constant

Table 3.2: *Guard syntax written in Backus-Naur form (BNF)*

Guards can be constructed by in terms of other guards; 'guardName' refers to a previously defined guard.

Two special guards exist for examining the progress in an orthogonal component: `IN` and `DONE`. `IN`(`stateName`) returns true if and only if `stateName` is an active state in a cluster or orthogonal component. The `IN` function can be used to guard outgoing transitions or to synchronize the components of an orthogonal state. `DONE`(`componentName`) returns true if and only if the specified component has completed its portion of a merge transition.

3.6 Response actions

Response actions are programmed responses to events. Each response action has a trigger event (for signalling when it executes) and a code segment (consisting of user-defined Tcl code). Section 3.7 illustrates how to specify a response action.

Any of the event types listed in Section 3.4 can be used to trigger a response action. In addition, two special events are defined for this purpose:

1. *Entry*: the code segment gets executed upon entrance to the state

2. *Exit*: the code segment gets executed just before departure from the state

Response actions execute from within a state. When a response action has completed execution, control returns to the same active state (unless, of course, the trigger was caused by an *Exit* event). Response actions do not cause the state to be exited and re-entered; the *Entry* and *Exit* events are not automatically executed as part of each response action.

The functions in Table 3.3, intended to represent common responses, can be referenced directly from within a response action's code segment. The functions are mostly straight-forward. The **mail** function is a convenience for sending email alerts, i.e. for reporting error conditions. The **record** function will store an entry in the output. The IAM software can be setup to output to either file or database. If database output is used, assigning the optional arguments an empty string ('') will cause a NULL value to be stored. By default, *value* and *description* contain an empty string.

get *sensorName* Returns the current value for the *Blackboard* sensor identified by *sensorName.*
getActive Returns a Tcl list containing the currently active states.
in *stateName* Returns 1 if *stateName* is an active sub-state in an orthogonal component, 0 otherwise.
mail *who title msg* Transmits an email to the destination address *who* with the subject line *title* and body *msg.* Returns nothing.
record *[value] [description]* Records an entry into the activity model output. All recorded values automatically include a timestamp, identification of the state that called **record**, and a listing of the currently active states. The optional parameters, *value* and *description*, are for user-defined customization; *description* often explains what *value* represents. Returns nothing.

Table 3.3: Functions calls useful within response actions

The finer points related to response actions:

- Scope. All response actions are executed at the global scope. This allows global variables to be used for communication between response actions of different states.
- Use of procedures. Tcl procedures can be defined for commonly occurring sequences of instructions. These procedures can then be executed as part of any response action. Subsections 3.7.1 and 3.7.4 describe how to create user-defined procedures.
- Atomicity. Once a response action starts to execute, it does not release the CPU until it has completed.

3.7 Mapping the model to a program

This section explains the procedure to map an *Activity Model* into the syntax of an *Activity Program.*

As an analogy, the first step in setting up a database is to model the manner that the data will be stored. Following this, the model is mapped into tables and relationships that the database tool can work with. Our methodology follows a similar path. First, a model is created for the operation of interest. Second, the model is mapped into syntax that is recognized by the software tool.

Section 3.7.1 outlines the format of any *Activity Program.* The subsequent sections detail more specifically how an *Activity Model* maps into the available functions.

3.7.1 Format of an Activity Program

An *Activity Program* can be viewed as a configuration file to setup the application-specific states, transitions, and actions being used. The basic format is displayed in Program 3.7.1. *Activity Programs* should abide by Tcl's syntax rules.

All *Activity Programs* begin with

```
#!/usr/bin/tclsh
package require iam 1.0
iam
```

The first line identifies that we are creating a Tcl script program. The second line allows access to the functionality implemented in our IAM package (assuming it has been installed on the target system). The `iam` command performs initialization, and should be executed before any other IAM function is used.

It is important to note that all of the upper-case headings (i.e. "`INPUT CONFIG:`", "`CONSTANTS:`", ...etc.) are required, even if a given section is empty. The headers themselves are function calls that perform further initialization. As an example, the `END` command turns execution control over to the IAM software.

The syntax of the remaining commands is given in Table 3.4. The following subsections will step through the use of these commands in more detail.

Program 3.7.1 Skeleton of an *Activity Program*

```
#!/usr/bin/tclsh
package require iam 1.0
iam

INPUT_CONFIG:
  connectTo computerName portNo

  # raw sensors
  streamRegister sensorName
  ...

  # derived/soft sensors
  createKS softSensorName KS_type {inputList} updateRate <args>
  ...

CONSTANTS:
  constant CONSTANT_NAME value

EVENTS:
  eset EventName eventDefinition
  ...

GUARDS:
  gset GuardName guardDefinition
  ...

PROCEDURES:
  proc ProcedureName {args} {
    # procedure body
  }

STATES:
  cluster <State1> {
    basic <State2> {
    }
  }

TRANSITIONS:
  transition {fromStates} {toStates} {EventNames} {GuardNames}

END
```

connectTo *hostname portNo* Establishes a connection with the computer identified by *hostname*, presumably executing a *Data Retrieval Server* application on port *portNo*. Returns nothing.
streamRegister *sensorName* Registers the IAM package to continuously receive updated values for the sensor identified by *sensorName*. Returns nothing.
createKS *softSensorName KS_type inputList updateRate [KS specific]* Initializes a new soft sensor, named *softSensorName*. The *KS_type* indicates the type of *Knowledge Source* that will be instantiated; currently `{trend, discretization}` types are supported. The *Blackboard* variable(s) that will serve as input to the KS are listed in *inputList* (a Tcl list). The *updateRate* parameter gives the number of input arrivals to be received before the *softSensorName* sensor is updated. The *KS_specific* argument is a Tcl list that is interpreted according to the *KS_type* used (see Section 3.7.2). Returns nothing.
constant *name value* Creates a constant named *name* and assigned value *value*. Returns nothing.
eset *name definition* Declares a new event type, with name *name* and definition *definition*. Returns nothing.
gset *name definition* Declares a new guard type, with name *name* and definition *definition*. Returns nothing.
basic *stateName definition* **cluster** *stateName definition* **component** *stateName definition* **orth** *stateName definition* Declares *stateName* as a {basic, cluster, component, orthogonal} type. *stateName* should be delimited by angle brackets, i.e. <aState>. The *definition* specifies response actions and/or sub-states (if it is not basic). The first sub-state defined for a component/cluster is its default starting state.
event *eventName definition* Declares *eventName* to be an event that triggers a response action. *eventName* should be delimited by angle brackets. The response code to execute is given by *definition*.
transition *fromStates toStates events [guards]* Specifies a transition. All four arguments are passed as Tcl lists. *fromStates* contains the state(s) being exited, *toStates* contains the state(s) being entered. States are described using absolute state names, in the format of Section 3.7.7. The ordering of the transition *events/guards* corresponds to the state ordering in *fromStates*. Returns nothing.

Table 3.4: Commands used within Activity Programs

3.7.2 Specifying input

All input comes from the *Data Retrieval Server.* Consequently, each IAM program must execute the `connectTo` function.

Sensors whose values are streamed from the *Data Retrieval Server* are registered with the `streamRegister` command. The *sensorName* argument serves a dual purpose. First, it is how the *Data Retrieval Server* refers to the sensor. Second, it is how the sensor will be referenced in the *Blackboard.* On arrival of a new sensor value from the DRS, the value is written to the corresponding *Blackboard* variable. The *Blackboard* variables are accessible to the *Activity Program.*

Inputs that are dynamically created are each specified with a `createKS` command. Each `createKS` command instantiates a *Knowledge Source* (see Section 3.3.2.3). The KS instantiation will automatically update a *Blackboard* variable named *softSensorName.* The *KS_ specific* argument has different meanings for each KS type:

1. `trend`. The Tcl list contains two parameters, both for configuring the storage of the data samples used to compute the trend of the sensor. The first parameter is an integer for the maximum number of data samples to store. Once the storage space is full, new sensor values will replace the oldest sensor values. The second parameter is an integer to indicate the minimum number of data samples to collect before calculating the first derived value. This is to ensure that the first few derived values are not biased due to a small sample size.

2. `discretization`. The Tcl list contains two parameters: a *divideList* and an *intervalList.* The *divideList* is a Tcl list of ascending sensor values that divide the input into a finite number of ranges. The *intervalList* is a Tcl list containing a name for each finite range. The sequencing order of the *intervalList* should correspond to the entries in the *divideList.* The *divideList* should have exactly one more entry than the *intervalList.*

3.7.3 Mapping events and guards

The `eset` and `gset` commands define all events and guards, respectively. Specifying an event assigns a name to an instantiation of one of the event types listed in Section 3.4. Specifying a guard assigns a name to one of the boolean expressions listed in Section 3.5. Note that the IN and DONE guards require their arguments to be specified in absolute format (see Section 3.7.7). Some examples include:

```
eset Unusual ThreshExceeded(aBlackboardVariable,>=,$MAX_READING,1)
eset Normal  ThreshExceeded(aBlackboardVariable,<=,$HIGH_READING,2)
eset TenSeconds Timeout("00:00:10")

gset Unstable IN(.OrthState.Bottom.aCluster.State_2)
gset HighValue aBlackboardVariable >= $HIGH_READING
```

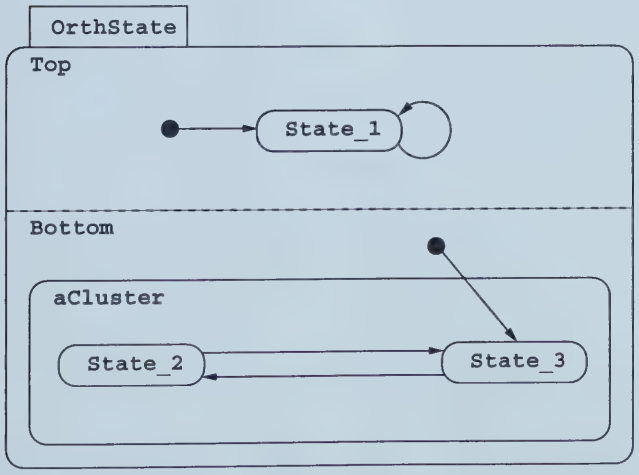
Once the events and guards have been defined, it is recommended to replace the numeric thresholds used therein with constants. A constant is a special type of variable that can be initialized (once) with a value but thereafter cannot be assigned a new value. In the above example, assume that `MAX_READING` and `HIGH_READING` have previously been defined as constants. Constants are created with the `constant` command, and positioned under the "`CONSTANTS:`" heading in the *Activity Program*.

3.7.4 Adding procedures

Procedures are useful when the same sequence of commands are shared between different response actions. Procedures are inserted using standard Tcl syntax, and can be inserted following the "`PROCEDURES:`" header.

3.7.5 Mapping states

Figure 3.11 illustrates how a hierarchy of states is translated into text[6]. `State_3` is defined before `State_2` because components and clusters identify their default states by listing them first. Figure 3.11(b) makes use of the four state-creation commands of Table 3.4: `basic`, `cluster`, `component`, and `orth`. Note that the *definition* of the state types is not optional; in cases where a basic state has no events the *definition* becomes '{}'. The program is set to ignore white space, so '{ }' is equivalent.



(a) Graphical Description

```
orth <OrthState> {
  component <Top> {
    basic <State_1> {
    }
  }
  component <Bottom> {
    cluster <aCluster> {
      basic <State_3> {
      }
      basic <State_2> {
      }
    }
  }
}
```

(b) Textual Description

Figure 3.11: Specifying states using text

[6]Our method of using text to describe the hierarchy of Statechart states is similar in idea to what is done by Lucas [23].

3.7.6 Mapping response actions

Response actions can be inserted inside the braces of a state's *definition.* The format of each response action looks:

```
event <EventName> {
  ResponseCode
}
```

The *EventName* is to be replaced with one of the events defined with the `eset` command; alternatively, it can also be set to one of the pre-defined event types {*Entry*, *Exit*}. The *ResponseCode* consists of arbitrary Tcl code. The functions listed in Table 3.3 are useable as part of the *ResponseCode.*

Continuing our previous example from Figure 3.11(b), let us assume we want to track the entrance to each state and component. Program 3.7.2 shows a modified version, with response actions that execute the `record` function.

Program 3.7.2 Figure 3.11(b) code with simple response actions

```
orth <OrthState> {
  event <Entry> { record }
  component <Top> {
    event <Entry> { record }
    basic <State_1> {
      event <Entry> { record }
    }
  }
  component <Bottom> {
    event <Entry> { record }
    cluster <aCluster> {
      event <Entry> { record }
      basic <State_3> {
        event <Entry> { record }
      }
      basic <State_2> {
        event <Entry> { record }
      }
    }
  }
}
```

3.7.7 Mapping transitions

All transitions are specified with the `transition` command, listed in Table 3.4. The transition command does error-checking to ensure that events/guards have previously been defined by the `eset`/`gset` command, and that the identified states exist.

Transitions are specified with absolute state names, which is the manner that an *Activity Program* recognizes each state. An absolute state name indicates all of the ancestors of a particular state. It

starts with a period and has a period between all ancestor states, in order, up to the state being described. For example, the absolute names of the states appearing in Figure 3.11 are:

```
.OrthState
.OrthState.Top
.OrthState.Top.State1
.OrthState.Bottom
.OrthState.Bottom.aCluster
.OrthState.Bottom.aCluster.State2
.OrthState.Bottom.aCluster.State3
```

The special transition types presented in Section 3.2.5 are for modelling purposes only. Guard branches, junction connectors, and diagram connectors exist to allow ideas and concepts to be understood easier. As illustrated by Figures 3.7, 3.8, and 3.9, these mechanisms can be equivalently represented with plain transitions. The conversion into plain transition format is necessary for specification with our tool.

A transition can have one or more source states and one or more destination states. Transitions entering (leaving) a cluster or basic state only require a single destination (source) state. However, transitions dealing with orthogonal states may require multiple destination/source states to be specified. An orthogonal state that is exited by a merging transition arrow needs to indicate multiple exit states. An orthogonal state that is entered by a splitting transition needs to specify the starting sub-states for more than one component.

The rule to remember is that transition triggers are from the perspective of the exiting state, so the number of events/guards in the transition command always match the number of exiting states. In most cases, only a single event and potential guard are expected. The exception is when a merge transition exits an orthogonal state, where each sub-state involved in the merge lists its own triggering event and optionally a guard.

When an orthogonal state transition has multiple sources/destinations, the sub-states are surrounded by curly braces and delimited by commas. For example:

```
.root.orthState.{comA.substate1,comB.substate2}
```

indicates an orthogonal state with two components (`comA` and `comB`), each specifying a sub-state (`substate1` and `substate2`, respectively).

Note that it is also possible to specify the substates of a sub-orthogonal state within another orthogonal state. For example:

```
.root.orthState.{secondOrth.{comA.sub1,comB.sub2},comC.sub3}
```

corresponds to the following absolute state names:

```
.root.orthState.secondOrth.comA.sub1
.root.orthState.secondOrth.comB.sub2
.root.orthState.comC.sub3
```

3.8 Example: truck torsion tubes

The intention of this section is to work through an example that demonstrates how *Activity Model* solutions can be constructed. The *Activity Program* for this example can be found in Appendix A.1.

3.8.1 Motivation for monitoring trucks

In the recent past, a number of Syncrude trucks required maintenance due to cracks in their torsion tubes. Each torsion tube runs the length of the truck much like the backbone of a person. A study was performed to measure the amount of stress placed on the tube components during normal operation. In this section, we propose an *Activity Program* that could have potentially helped analyse the results of that experiment.

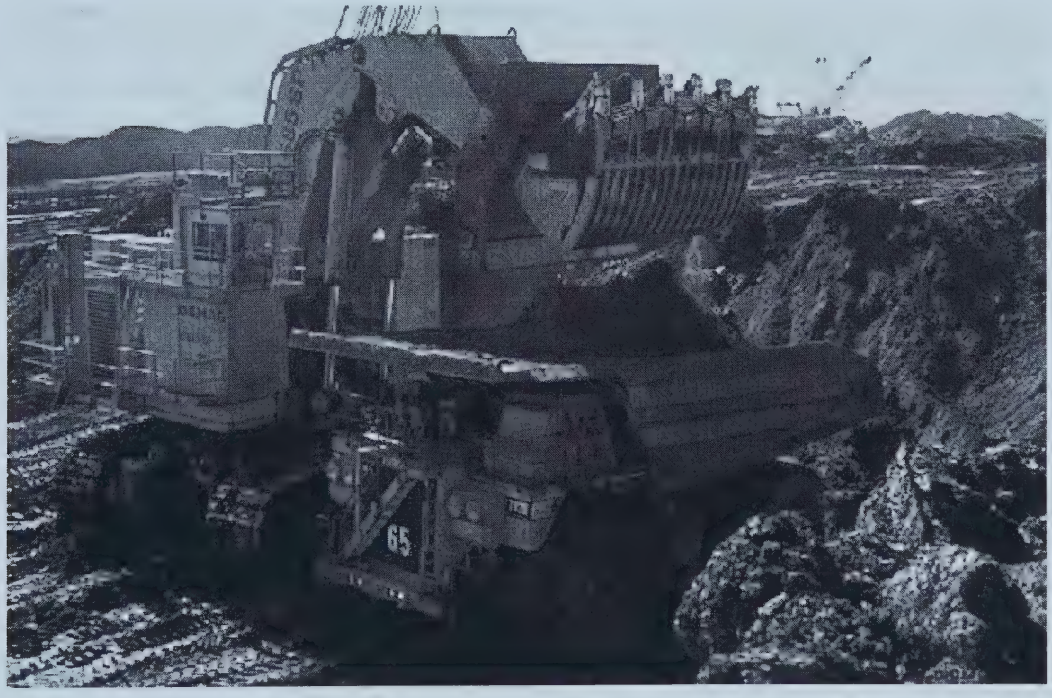


Figure 3.12: Caterpillar 793B haul truck being loaded by a Demag shovel

Contextual information that would be of interest includes:

- *Status.* Different pressures are placed on the truck depending on whether it is engaged in loading, driving, or dumping activities. As well, driving with a full load adds more strain than driving with an empty load.
- *Turning.* Each turn shifts the weight of the truck, affecting the torsion tubes

- *Truck angle during loading.* Any unevenness in the ground that causes flexure in the truck frame may be amplified or reduced during loading [14].

- *The truck box after dumping.* If the box is still raised when the truck starts moving away after dumping, the box puts additional pressure on the torsion tubes.

3.8.2 Model design

To monitor the *Status* of the truck, we decide to use four states: `Loading`, `Driving_With_Load`, `Dumping`, and `Driving_Without_Load`. We know, through knowledge of the process we are studying, the sequencing of these states.

We want to be simultaneously monitoring the *Turning* of the truck. We elect to monitor the *Turning* from a separate orthogonal component on the assumption that the tire direction is of interest regardless of whichever *Status* state is active at the time. In particular, we can speculate that "dry turning" while the vehicle is parked might place more stress on the torsion tubes than "normal turning" while the vehicle is moving.

Our design so far is drawn in Figure 3.13.

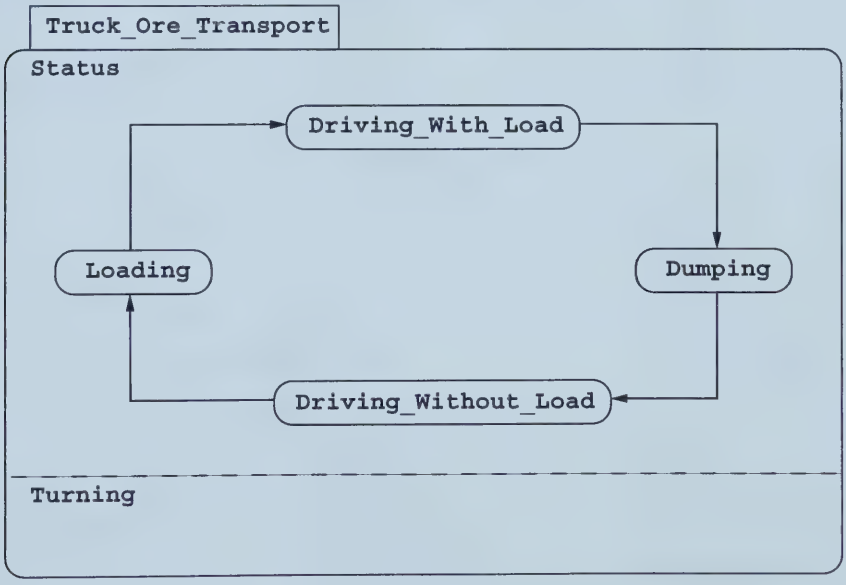


Figure 3.13: First design

To track *Turning* we assume the existence of a sensor that measures the angle of the wheels. We'll refer to this sensor as **wheelAngle**. For the purposes of our study, let's assume it is sufficient to keep track of whether the truck is turning-left, going straight, or turning-right. We decide on a discretization-type *Knowledge Source*, call it **wheelDirection**, for classifying the **wheelAngle** value into one of the expected categorizations.

Turning can be implemented with a single state: `DirectionMonitor`. Each time there is a change in the **wheelDirection** variable, the `DirectionMonitor`'s *Entry* response action records the new direction.

We will now specify the events that trigger transitions in the `Status` component. *Moving* seems an appropriate trigger to switch from stationary states (`Loading`, `Dumping`) to the driving states. *Moving* is an event based on the truck's velocity exceeding some nominal speed. To transfer from `Driving_With_Load` to `Dumping`, we use the *Box Raised* event. Our argument is simply that once the box is raised, the contents of the truck are being dumped (regardless of whatever else is going on in the truck). *Box Raised* assumes we have access to a sensor that measures the angle of the box; we refer to this sensor as **boxAngle**. The transition from `Driving_Without_Load` to `Loading` is triggered by the *Weightometer_Jump* event. We assume the truck is equipped with a weightometer that measures its load. When a significant increment occurs in the weightometer reading, we conclude that a new load was dumped onto the truck. We use a trending *Knowledge Source* called **weightTrend** to detect significant changes in the weightometer's reading. In contrast to using a fixed weightometer threshold, the *Weightometer_Jump* event works even if tar has become adhered to the truck's box. Figure 3.14 represents our model at this point.

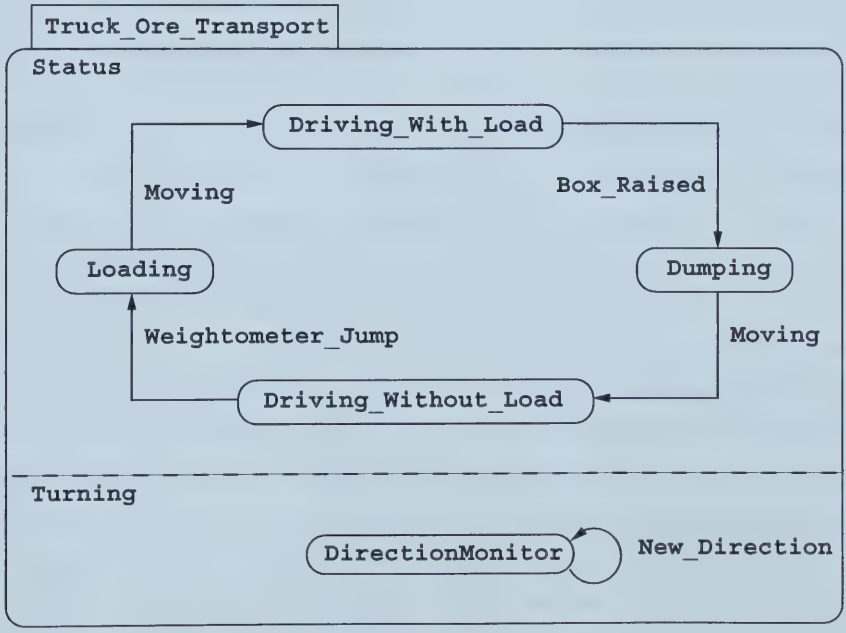


Figure 3.14: Second design

Next we will incorporate the *"Truck Angle During Loading"* item.

We assume a sensor called **truckAngle** that reveals the truck's tilt. After each load the truck receives, we want to record the latest **truckAngle** measurement. The argument here is that the

truck angle may vary due to the distribution of each load that it receives. As well, sometimes when the shovel is busy digging the truck operator will maneuver slight "corrections" in order to get closer to the shovel.

We subdivide the `Loading` state into `Waiting` and `ReceivingLoad` states. `ReceivingLoad` represents the time frame during which material is landing in the truck's box. `Waiting` represents the time frame during which the shovel is swinging away, scooping up a new load, and swinging back. The *Weightometer_Jump* event is re-used for indicating a transition from `Waiting` to `ReceivingLoad`. The reverse transition from `ReceivingLoad` to `Waiting` is triggered by the *Weightometer_Stable* event – fired when the weightometer measurement ceases to increase for some duration.

It is time to consider the issue of the *"Truck box after dumping"*. We insert a new state `Driving_With_Box_Raised` to the `Status` component, and introduce two guards: Box_Down and Box_Up. The status of the box upon exiting the `Dumping` state determines whether the next state is `Driving_With_Box_Raised` or `Driving_Without_Load`. Assuming we first traverse through the `Driving_With_Box_Raised` state, we can compare its entrance timestamp against that of `Driving_Without_Load` to determine the duration that the box was raised for.

We now start to think about how our model will behave in unexpected circumstances. For example, what if the truck repositions itself during loading and inadvertantly triggers the *Moving* event? (In theory, this should not be a problem if the *Data Retrieval Server* filters out noise and if the *persistence* is set correctly). The false alarm would fool our system into thinking it was in the `Driving_With_Load` state. We can recover from this situation by adding a transition from `Driving_With_Load` to `Loading` on the occurrence of the next *Weightometer_Jump* event.

Finally, we add default arrows to the components. The `Turning` component's default state was easy to pick; for the `Status` component, we arbitrarily selected a state. The `Status` component will correct itself within a cycle regardless which state is the starting state.

On a debugging note, we would like to verify that the specified event transitions are correct. One easy validation check is to ensure our model doesn't get stuck in any particular state for an exceedingly long time. We define a *Model_Stuck* event for recognizing an unacceptable delay. For the sake of argument, we superficially set the unacceptable delay to be four hours. A more precise value requires a history of the durations of each state; an unacceptable delay would be one that surpassed the maximum duration recorded in the history by a sufficient amount. We selectively choose to monitor for the *Model_Stuck* event from the five key `Status` states: `Loading`, `Driving_With_Load`, `Dumping`, `Driving_With_Box_Raised`, and `Driving_Without_Load`. If the `Model_Stuck` event fires, it executes a procedure that emails a warning message to the appropriate party.

3.8.3 Model review

We will now review how our proposed *Activity Program* addresses the points raised in Section 3.8.1.

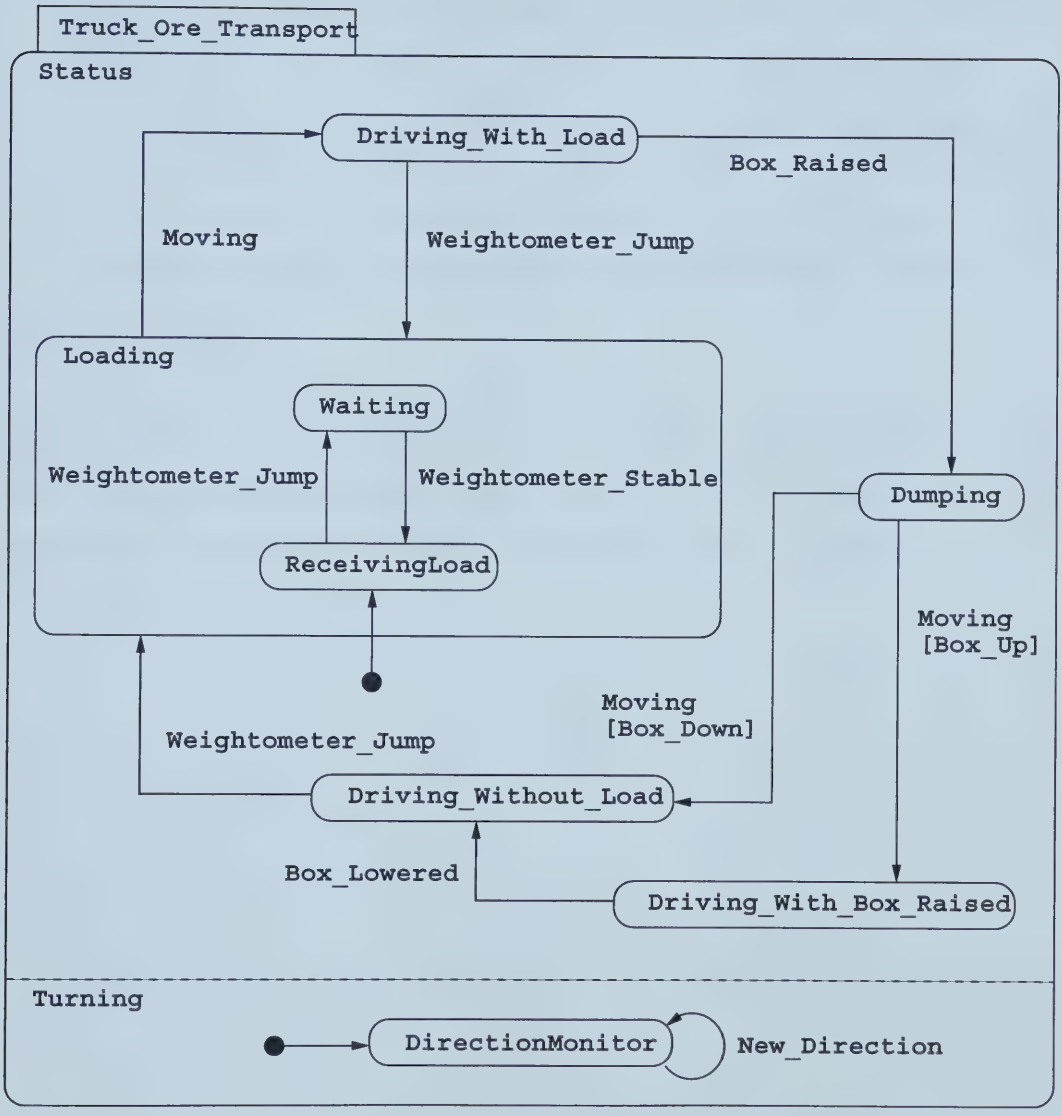


Figure 3.15: Final design

- *Status.* The different driving modes are identified through the following four states: `Loading`, `Driving_With_Load`, `Dumping`, and `Driving_Without_Load`. Each time one of these states is entered a response action is invoked to record the state's name and the timestamp.

- *Turning.* Each time the steering direction changes among turning-left, going straight, turning-right, the `DirectionMonitor` state is re-entered. The event of entering the `DirectionMonitor` state invokes a response action that records the new wheel direction and a timestamp.

- *Truck angle during loading.* Everytime the `Waiting` state is entered, which signifies a new load was just received, the most recently measured truck angle is recorded with a timestamp.

- *The truck box after dumping.* The duration of time spent in the `Driving_With_Box_Raised` state indicates the amount of time where the truck is moving with its box still raised.

3.9 Summary

We have constructed a tool that makes it easy to create programs that monitor equipment. We propose a two-step procedure. The first step is to create an *Activity Model* – a diagram that specifies how the equipment is to be monitored. The second step is to convert the *Activity Model* into a corresponding *Activity Program* that can be executed with the support of our toolkit.

Chapter 4

Experiment and Results

This chapter provides an example usage for our IAM tool. We develop an *Activity Model* of Syncrude crushers in the winter, motivate the significance of the model, and graph the output of the corresponding *Activity Program*. Complete syntax for the *Activity Program* is included in Appendix A.2.

4.1 Problem setup

4.1.1 Syncrude crushers in the winter

The role of the double-roll crushers (DRC) at Syncrude is to grind truck loads of oil sand into smaller particles. These smaller particles are then carried away on a conveyor belt and deposited in a surge pile. The surge pile serves as a storage bin that feeds oil sand into cyclofeeders. Cyclofeeders combine the oil sand with water so that it can be hydrotransported through pipes into the bitumen extraction plant. The extraction facilities expect the incoming slurry to arrive at a constant rate. It is therefore critical that the surge pile does not become empty, and in turn for the crushers to be operating at expected levels.

Figure 4.1 diagrams the relationship between the key crusher components. The solid arrows indicate the travel direction of oil sand.

In the winter months, oil sand can solidify into boulder-like lumps that present a problem for the crushers. Large oil sand lumps can cause the crusher to *jam* or *plug*. A jam is when the grinders stall, requiring the assistance of the operator. A plug is when the grinders are slowly working away at a lump, with a buildup of oil sand on top (Figure 4.2).

Lumps are dealt with in the following ways:

- If a human operator takes notice of a large lump progressing up the apron feeder, the apron feeder is manually disabled once the lump falls into the crusher chute. This is done on the assumption that the lump can be processed without jamming the crusher. If the apron feeder is not stopped the additional oil sand increases the likelihood of the crusher jamming or becoming plugged.

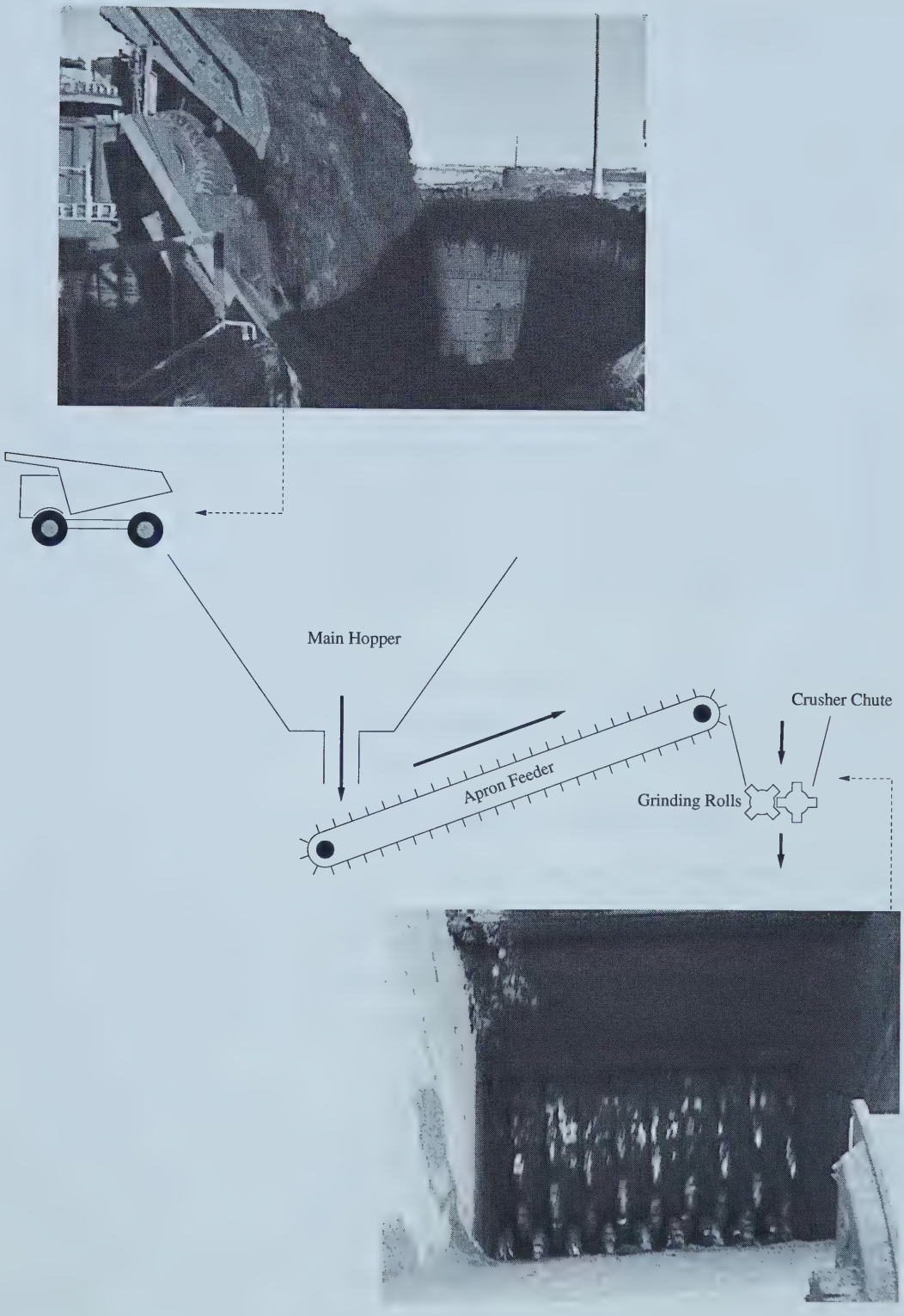


Figure 4.1: Arrangement of crusher components

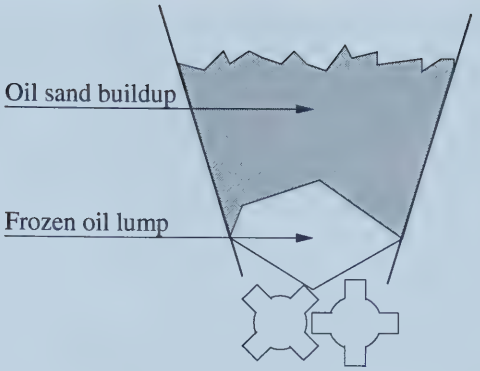


Figure 4.2: Plugged crusher

- If a lump stalls the rolls, an automatic response disables the rolls and the apron feeder. A human operator resolves the conflict. One judgement may be to restart the rolls and try again. A second approach is to first reverse the roll direction, dislodging the lump, and then re-attempt to grind it. In extreme cases, where these approaches repeatedly fail, a back hoe excavator is required to manually remove the lump from the crusher chute. An example of this latter case is shown in Figure 4.3.

- If an undetected lump plugs the crusher chute (before it has been disintegrated) the resulting rise in the chute level trips an alarm that suspends the apron feeder. Once the lump has been processed, the apron feeder is re-enabled.

- Undetected lumps will occasionally grind through the crusher without alarm.

Before we go forward, we would like to express an operational difference between the apron feeder and the rolls. The apron feeder can be turned on and off easily, and is automatically disabled whenever it is economical to do so. In contrast, turning the rolls on and off wears them down quite a bit. Very little trips the crusher rolls except for motor protection, belt slips, or gear box problems [8]. Consequently the rolls are always left running whenever they can be, even if they are not grinding anything.

4.1.2 Purpose of study

We create an *Activity Program* that executes on 2 months of previously collected data. A description of the input data is given in Section 4.1.3.

The purpose of any IAM post-analysis is to determine how much time is spent in each of the outlined states. With respect to crusher monitoring, an *Activity Program* can be helpful with the following:

Figure 4.3: An operation interruption caused by a frozen oil sand lump

A. Downtime Activities. What activities is the crusher engaged in when it is not operating? Why is the crusher not operating?

B. Crusher Utilization. This statistic becomes more accurate when the time spent actively crushing lumps is factored in.

The next two subsections elaborate on these goals.

4.1.2.1 Downtime activities

One of Syncrude's data-collecting systems is dubbed NMET, short for *North Mine Event Tracking.* Among other things the NMET system records lump occurrences. However, the NMET lump data is manually entered by humans and only includes lumps that take more than 5 minutes to disintegrate.

Another use of the NMET system is to allow the operators to record why the crusher is inactive, during moments of downtime. But these descriptions tend to be brief and are not consistent between operators.

We therefore suggest an *Activity Program* to provide an automatic and more refined analysis of what is happening while a crusher is down. We focus on the following issues:

- Is the NMET lump data accurate? Is it worthwhile to additionally record lumps that take less than 5 minutes to process?

- How much time is spent trying to unjam lumps with the reverse rolls strategy? (This is hard on the equipment)
- What is the average delay following a jam before action is taken to correct the problem?
- What percentage of crusher downtime is attributable to lumps?

The overall goal is to keep Syncrude knowledgeable about how the crusher is being operated, particularly during lump activity.

4.1.2.2 Crusher utilization

Syncrude expects each crusher to be operating a certain percentage of the time. Failure to reach these targets causes investigations to be launched to determine which problems need to be corrected. Some of this investigation work can be avoided if in fact the crushers are operating longer than the accounting methods show.

The current accounting system reports the crusher as being ineffective when the apron feeder has been de-activated. This is appropriate for summer operation, where the apron feeder is only turned off when there is no material to transport from the main hopper. The question is whether this criteria is equally appropriate in the winter months, considering that a significant portion of time is spent processing frozen lumps with the apron feeder off. Classifying the crusher as ineffective when lumps are being processed is misleading and may even bias the accounting for these months.

On this front, our objective is to learn the proportion of time whereby the crusher is processing lumps with the apron feeder off. If this proportion of time is significant, it suggests modifying the accounting methods to include lump processing times.

4.1.3 Input data

Our study focuses on Syncrude's 757 double-roll crusher. Data was available for the months of January and February of 2000, with accuracy as listed in Table 4.1. We assume all of the input sources to be time synchronized.

This section merely points out which inputs were selected, given the data available. How these inputs are used within an *Activity Model* is the subject of Section 4.2.

The *Real-Time Journal* (RTJ) alarms are obviously the most precise information points. However, we found that most alarms report on quite specific instances (i.e. high oil temperature) and their coverage was not broad enough to create a meaningful self-contained model. Two alarms did prove useful, these being the indications of when the operator manually set the roll direction to be either forward or reverse.

The *Process Information* (PI) system provides us with records of the apron feeder speed, the roll speeds, and the main hopper level. We do not attempt to filter noisy data points; the PI data is

Name	Description	Accuracy
RTJ	Syncrude's *Real-Time Journal* system keeps a record of each alarm occurrence.	Nearest second
PI	PI stands for *Process Information*, a Syncrude wide system that records over 84,000 data points per minute. A necessary consequence of the amount of recorded data is that compression mechanisms are used.	Nearest minute
NMET	Syncrude's *North Mine Event Tracking* system requires crusher operators to log all lumps having a processing duration of longer than five minutes; shorter lump periods are not recorded.	Nearest minute

Table 4.1: Input sources of crusher data

already compressed. Note that the PI data is occasionally less frequent than once-a-minute because marginal differences from one minute to the next are not logged.

The 757 crusher rolls are numbered C08 and C09. Our observation is that these rolls operate in parallel – they are either both on or both off. For brevity, we will only make use of the C08 roll.

The *North Mine Event Tracking* (NMET) lump data is compared against the lump occurrences detected by our model.

4.1.4 Scheduled maintenance

We did not want our model to mistakenly classify periods of scheduled maintenance as being lump incidents. The input data was manually filtered to identify the maintenance shutdowns. We created a text file that contains the start/stop times of these shutdowns. The *Activity Program* perceives this file as a "sensor" that indicates when maintenance begins and when it ends.

4.2 Solution design

4.2.1 Partial IAM model

This section introduces the key states of our model, as shown in Figure 4.4. Section 4.2.2 fills in the details of this design, and adds the necessary transition labels.

The top-level consists of an `Operating` state and a `SchedMaint` state. This breakdown cleanly separates the periods of scheduled maintenance from the rest of the model.

Within the `Operating` state, the topmost component is set aside to show the NMET lump data. The name `R_Lump` is short for "Recorded Lump". The bottom component contains our automatic lump detection strategy. The split into two separate components will allow us to directly compare the amount of lump downtime that each observes.

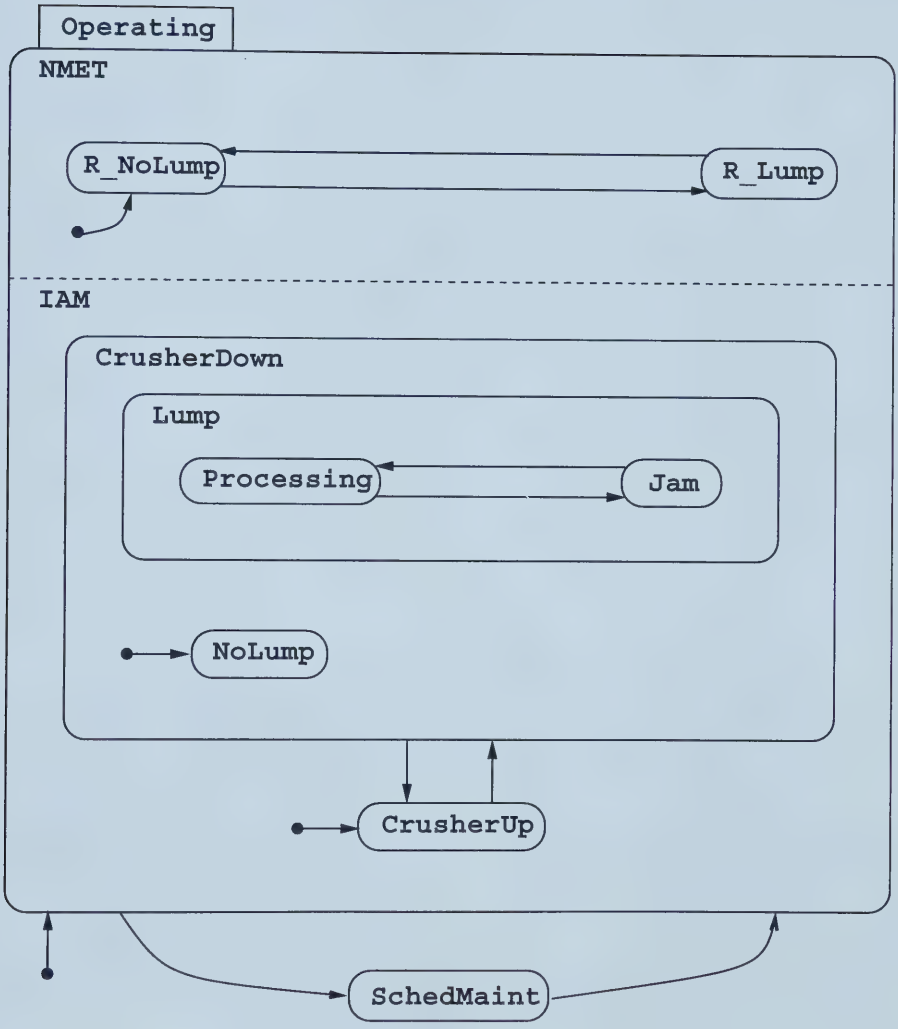


Figure 4.4: Key states of crusher model

The sub-states `CrusherUp` and `CrusherDown` are intended to reflect Syncrude's current classification strategy: the crusher is operational when the apron speed > 30%, and is inoperational otherwise.

The sub-states of `CrusherDown`, `Lump` and `NoLump`, classify whether or not a particular downtime is the result of a lump. Within the `Lump` state, whenever the rolls are working at proficient levels the `Processing` state is active. Otherwise the `Jam` state is active. The time spent in the `Processing` state is the time we suggest could be added to Syncrude's crusher utilization statistic.

4.2.2 IAM model

Our *Activity Program* executes according to the design in Figure 4.5. The meanings of the events and guards used are presented in Table 4.2. The complete code can be found in Appendix A.2.

Inside Table 4.2, 'ApronFeeder', 'RollSpeedC08', 'PileLevel', and 'HopperLvl' refer to the PI data for the apron feeder speed, the roll speed, the surge pile height, and the main hopper level, respectively. The 'Maintenance' is manually set to toggle between 0 and 1 at the correct moments to exclude scheduled maintenance periods from our results. 'LumpSignal' has a non-zero value during NMET lump instances, and a zero value otherwise. 'C08_Forward' and 'C08_Reverse' change value whenever an alarm reveals the operator manually set the roll direction to be forward or reverse.

Before showing where the thresholds in Table 4.2 evolved from, let us throw light on our choice of states and transitions.

Events		Guards	
apronOff	ApronFeeder ≤ 30	HiHopper	HopperLvl ≥ 50
apronOn	ApronFeeder > 30	HiSurge	(PileLevel ≥ 75)
endMaint	Maintenance ≤ 0		AND (HopperLvl ≥ 50)
forwardRolls	Change(C08_Forward)	LowHopper	HopperLvl < 50
hiHopper	HopperLvl ≥ 50	NoOverride	NOT IN(`Rev`)
lumpSignalOff	LumpSignal ≤ 0	NoReason	(PileLevel < 75)
lumpSignalOn	LumpSignal > 0		AND (HopperLvl ≥ 50)
reverseRolls	Change(C08_Reverse)	RollsOff	RollSpeedC08 < 90
rollsOff	RollSpeedC08 < 90	RollsOn	RollSpeedC08 ≥ 90
rollsOn	RollSpeedC08 ≥ 90		
startMaint	Maintenance > 0		

Table 4.2: Events and guards of Figure 4.5

4.2.2.1 The states

Table 4.3 reviews the significance of the states identified in Figure 4.5. The reader may notice sub-states have been added to three of the `CrusherDown` states: `Processing`, `Jam`, and `NoLump`. We will discuss each addition in turn. Let us first point out that if the `CrusherDown` state is active then by definition the apron feeder is off.

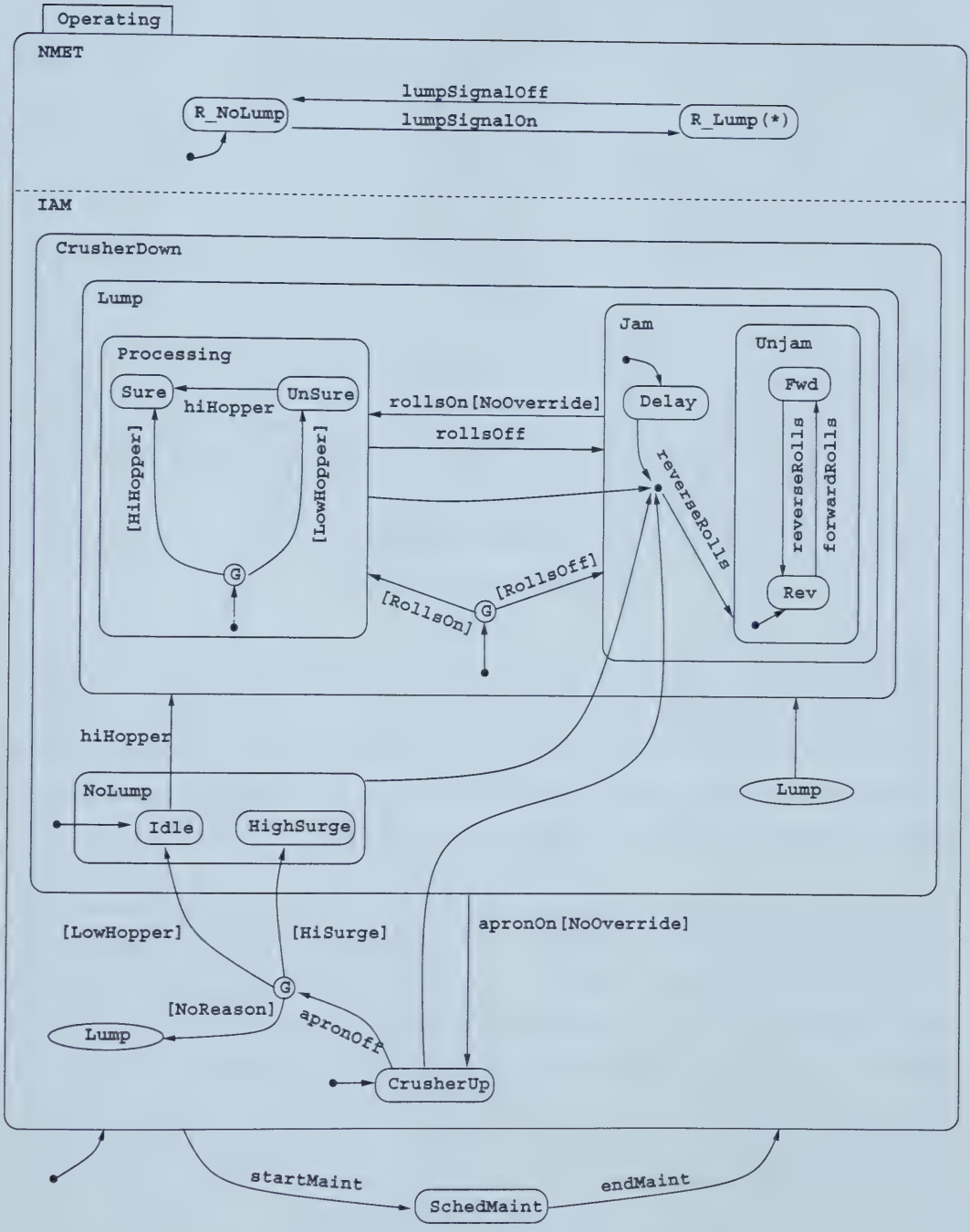


Figure 4.5: Crusher model

CrusherDown	The crusher is considered inoperative by current Syncrude standards.
CrusherUp	The crusher is operating normally.
Delay	A lump has been detected; the crusher has automatically been placed in a standby mode. This state represents the delay between the time that the alarms have tripped and the time when a human operator is present to deal with the lump.
Fwd	During Unjam, the rolls are moving forward.
High Surge	The apron speed has been reduced because of a high surge pile.
Idle	There is nothing to process; the main hopper has been depleted.
Jam	The system is stuck. Crusher output is virtually nil.
Lump	The apron feeder is off due to a lump.
NoLump	The apron feeder is off, but the cause is not attributed to a lump
Operating	No scheduled maintenance is being performed.
Processing	A lump is being grinded through.
R_Lump	The NMET data indicates a lump.
R_NoLump	The NMET data does not indicate a lump.
Rev	During unjam, the rolls are moving in reverse.
Sure	Within Processing, high confidence.
SchedMaint	Scheduled maintenance is being performed.
Unjam	A lump is being dealt with by the operator. The rolls are either moving in reverse, or are accelerating in the forward direction. Either way, the rolls are not effectively "processing" ore.
UnSure	Within Processing, low confidence.

Table 4.3: States in Figure 4.5

Our model does not have an automatic way of knowing when a lump has completed processing. We assume a lump has been cleared when the apron feeder speed rises again. However, the apron feeder is optimized to only operate when there is material in the main hopper. If the main hopper happens to be near-empty after a lump has been dissolved, the apron speed will not be re-activated until a new truck load arrives. Such instances may inflate the time spent in Processing higher than it should be, with the tradeoff of less time spent in Idle.

Our model deals with this by creating two sub-states within Processing: Sure and UnSure. Sure represents the instances where a lump occurs when the main hopper still contains more material. UnSure represents cases when a lump occurs with a near-empty main hopper. While within the Sure state, if there is any delay in re-activating the apron feeder, it cannot be attributed to a lack of material for it to transport. The Sure state gives a lower boundary on the amount of time spent processing lumps, when combined with the UnSure state we get an upper boundary.

The Jam state is subdivided into its constituents: a Delay state for measuring the length of time until an operator addresses the problem, and an Unjam state which keeps track of the proportion of time spent with the rolls moving forward and in reverse.

The NoLump state is supplemented with two sub-states: Idle and HighSurge. Both are alternate reasons, other than a lump, on why the apron feeder may be disabled. Idle is in reference to a main hopper that does not have enough material for the apron feeder to transport. HighSurge is in

reference to a downstream surge pile level exceeding 75%, and the apron speed being automatically reduced [9]. This reduction is to slow the input into the surge pile and prevent its overflow.

4.2.2.2 The transitions

This subsection reveals the logic underlying the transitions in Figure 4.5. The transitions are assigned labels in Table 4.4.

Label	State Transfer	Event	Guard(s)
t1	`CrusherDown→CrusherUp`	apronOn	NoOverride
t2	`CrusherUp→Sure`	apronOff	NoReason, RollsOn, HiHopper
t3	`CrusherUp→UnSure`	apronOff	NoReason, RollsOn, LowHopper
t4	`CrusherUp→Delay`	apronOff	NoReason, RollsOff
t5	`CrusherUp→Idle`	apronOff	LowHopper
t6	`CrusherUp→HighSurge`	apronOff	HiSurge
t7	`CrusherUp→Rev`	reverseRolls	
t8	`Delay→Rev`	reverseRolls	
t9	`Fwd→Rev`	reverseRolls	
t10	`Idle→Sure`	hiHopper	RollsOn, HiHopper
t11	`Idle→UnSure`	hiHopper	RollsOn, LowHopper
t12	`Idle→Delay`	hiHopper	RollsOff
t13	`Jam→Sure`	rollsOn	NoOverride, HiHopper
t14	`Jam→UnSure`	rollsOn	NoOverride, LowHopper
t15	`NoLump→Rev`	reverseRolls	
t16	`Operating→SchedMaint`	startMaint	
t17	`Processing→Rev`	reverseRolls	
t18	`Processing→Delay`	rollsOff	
t19	`Rev→Fwd`	forwardRolls	
t20	`R_Lump→R_NoLump`	lumpSignalOff	
t21	`R_NoLump→R_Lump`	lumpSignalOn	
t22	`Schedmaint→Operating`	endMaint	
t23	`UnSureSure`	hiHopper	

Table 4.4: Transitions in Figure 4.5

The arrow from `CrusherDown` to `CrusherUp` (transition t1) is perhaps the most subtle transition of them all. It is easy to get caught up in the sub-state workings of `CrusherDown` and forget that the aforementioned transition applies to all the sub-states. There is a NoOverride guard attached to this transition to help protect against false triggerings. The guard stops us from leaving the `CrusherDown` state if rolls are currently running in reverse (meaning that a lump is being dealt with).

From `CrusherUp`, a drop in the apron speed can be attributed to three things: a low hopper level, a high surge pile, or a lump. Each of these is represented by a guard branch emanating from the *apronOff* event. If the hopper is low, we accredit the event as the result of a lack of material to process (transition t5). Otherwise, if the surge pile alarms are in effect, we assume this is the cause of the apron speed reduction (transition t6). Finally, if none of the above criteria are satisfied we transfer into the `Lump` state (transitions t2-t4).

There is one other way of transitioning from `CrusherUp` to `CrusherDown`: if the rolls are reversed. When the *reverseRolls* alarm (transitions t7) occurs prior to the apron speed dropping (transitions t2-t6), this is just an indication that the compression procedure has delayed the update of the apron speed value. Clearly the roll speed must drop to zero before it can be moved in reverse. And the apron speed is automatically reduced when the rolls are off.

It is possible for our model to misclassify a lump instance as `NoLump`. For example, a lump may occur at the end of the processing of the main hopper contents; in this case our model would incorrectly follow transition t4 from `CrusherUp` to `Idle`. As well, it is entirely possible for a lump problem to coincide with a high surge tank. In the case of a misclassification, we rely on the *reverseRolls* event (transition t15) to reposition us inside the `Lump` state.

We also support a transition (t10-12) from `Idle` into `Lump` if the hopper level rises without the apron feeder being activated. If the apron feeder was to be activated, transition t1 would instead be followed into state `CrusherUp`. Note that transition t1 departs from state `CrusherDown` and therefore has priority over all transitions contained therein, including t10-12.

The remaining transitions should be self-explanatory.

4.2.2.3 The thresholds

This subsection explains the threshold values that appear in Table 4.2. Where helpful we will again refer to the transition labels of Table 4.4.

There are four threshold values that need explaining:

- What apron feeder speed indicates the apron feeder has been disabled?
- What surge pile levels can affect the apron feeder?
- What hopper level indicates the apron feeder should be working?
- What roll speed indicates that lump material is being processed?

When the apron feeder speed drops below 30% of its maximum speed, we assert that the apron feeder has been disabled. This is the same threshold used by the current Syncrude accounting system. This 30% threshold relates to the *apronOn* and *apronOff* events (transitions t1-6).

As mentioned above, an alarm is issued when the surge pile levels exceed 75%. The actual set point can be configured by the operator, however our study assumes the default value (75%) is always used. This threshold pertains to transitions t2-4 and t6.

We must quantify a threshold value that decides if the hopper level is "low." The moment the apron speed crosses beneath the 30% barrier, the hopper level helps determine how to classify the event. A low hopper level brings us to the `Idle` state (transition t5), while a high hopper level takes us either to the `HighSurge` (transition t6) or the `Lump` state (transitions t2-4).

Lets consider that the main hopper has a 700 tonne capacity, and that the average truck carries approximately 350 tonnes. Immediately after a truck load the main hopper contains material to be processed by the crusher; implying the apron feeder should be working, perhaps increasing in speed. A drop in the apron speed below 30%, when the hopper level is above 50%, cannot be attributed to an empty/low hopper. We therefore set 50% as our dividing threshold on the hopper level.

We have taken into account that 50% may be too high of an estimate; the automated signal that disables the apron feeder is probably triggered on a lower hopper level. However we note that if our threshold is in error it is biased against recognizing a lump, in favour of recognizing `Idle` time. This is appropriate given that one of our goals is to reveal a significant portion of time dealing with lumps.

The hopper level is also used to discriminate between `Processing`'s `Sure` and `UnSure` sub-states. The 50% threshold is generous towards the `UnSure` state, ensuring the `Sure` state remains a conservative measure of lump processing time.

Next we must discriminate a value for the roll speed that sets "rolls on" apart from "rolls off." When the `Lump` state is active, the roll speed decides between the `Processing` and `Jam` sub-states. Figure 4.6 graphs the empirical probability of each value that the roll speed takes on during January and February, excluding the maintenance times. Most commonly the rolls are either completely off or they are running at top speed. The intermediate speeds appear to represent moments of acceleration or de-acceleration between the two extremes.

We apply the logic that the roll speed is only reduced when the rolls are having difficulty grinding something. Therefore anytime the rolls are not fully operational the throughput is likely to be low. Based on Figure 4.6, it is fair to say that normal operating conditions are when the RPM is higher than 90. Anything lower is classified as "rolls off".

4.3 Results and discussion

Scripts were written that merged the input data (Section 4.1.3) together, along with the maintenance input (Section 4.1.4). The resulting file was then inputted into the IAM software, which ran according to the model specification of Figure 4.5. The resulting time breakdown between states is graphed in Figures 4.7 and 4.8. Table 4.5 contains a detailed list of the same data. The column marked '%' gives the utilization of each `Operating` state, defined as: state duration / operating time.

The first thing to notice is that the time spent in `CrusherUp` is comparative to that of `CrusherDown`. This alone indicates that it is worthwhile to further study how to increase the uptime of the crusher, if larger output is the goal.

Another interesting finding is the comparison between `R_Lump` and `Lump`. `R_Lump` represents the NMET reported lumps, whereas `Lump` represents the IAM detected lumps. For both months, the time spent in the `Lump` state is more than double that of `R_Lump`. This finding suggests that shorter duration lumps (less than 5 minutes) may be more influential in longterm crusher behaviour than

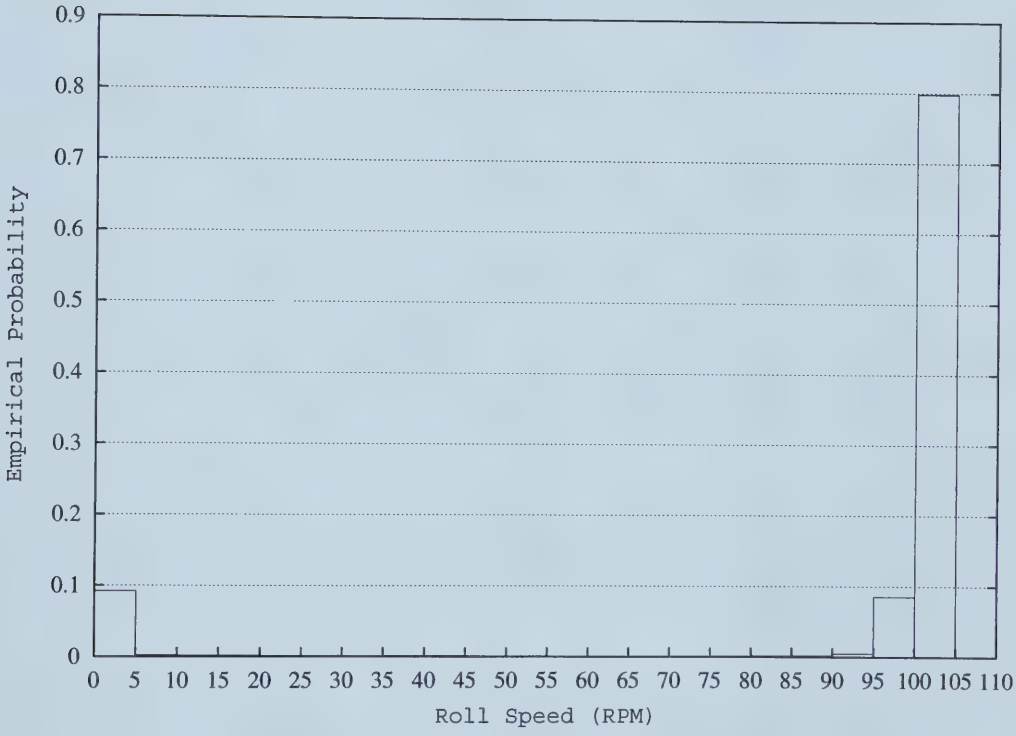


Figure 4.6: Histogram of Roll Speed values to the nearest 5 RPM

State	*January* Minutes	%	*February* Minutes	%
SchedMaint	3,054		3,688	
Operating	41,586		38,072	
NMET				
R_Lump	4,425	11	2,995	8
R_NoLump	37,161	89	35,077	92
IAM				
CrusherUp	22,915	55	23,814	63
CrusherDown	18,671	45	14,258	37
Lump	9,071	22	5,699	15
Processing	3,872	9	2,099	6
Sure	3,723	9	1,996	5
UnSure	149	0	103	0
Jam	5,199	13	3,600	9
Delay	437	1	645	2
Unjam	4,762	11	2,955	8
Fwd	3,045	7	1,846	5
Rev	1,717	4	1,109	3
NoLump	9,600	23	8,559	22
Idle	8,686	21	7,812	21
HighSurge	914	2	747	2

Table 4.5: Time spent in each state

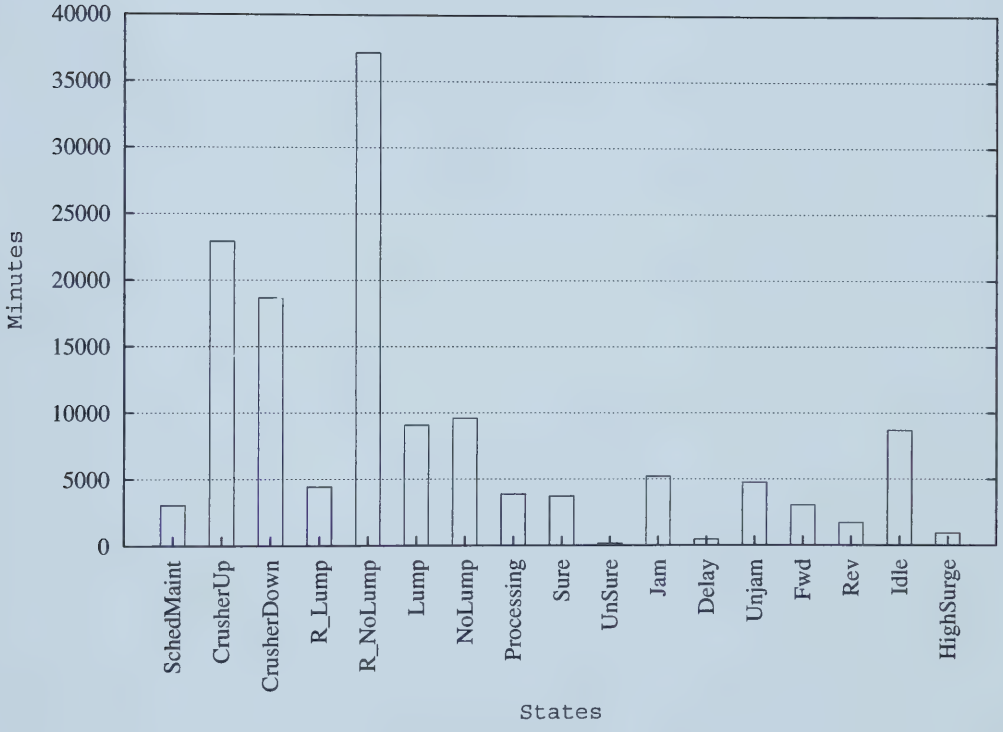


Figure 4.7: Total time within each state, January 2000

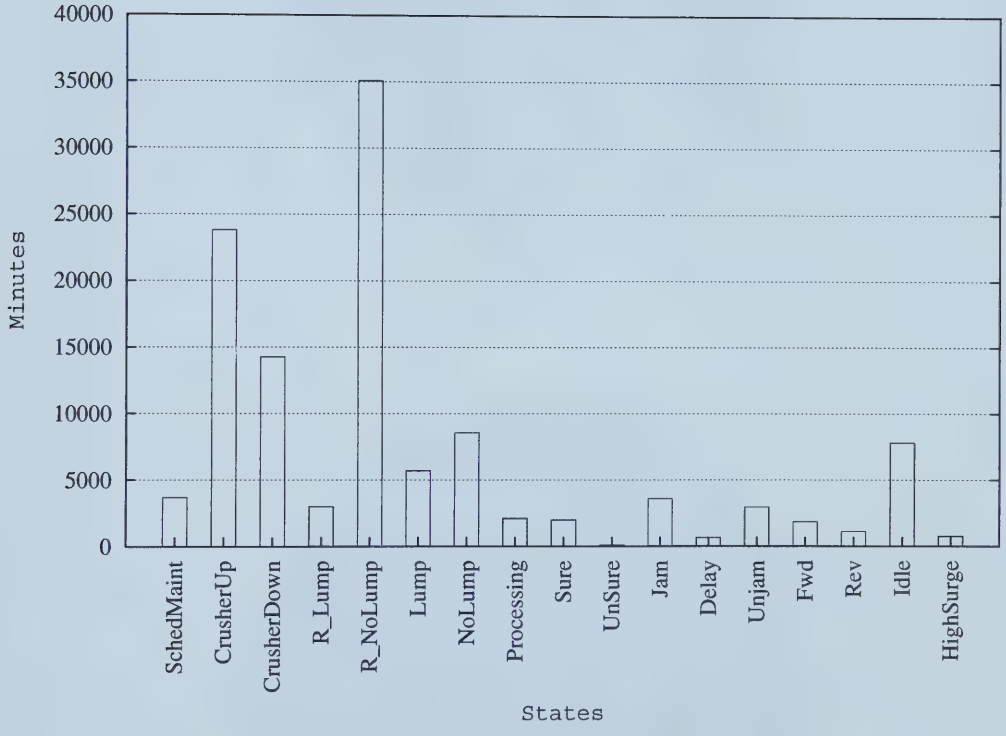
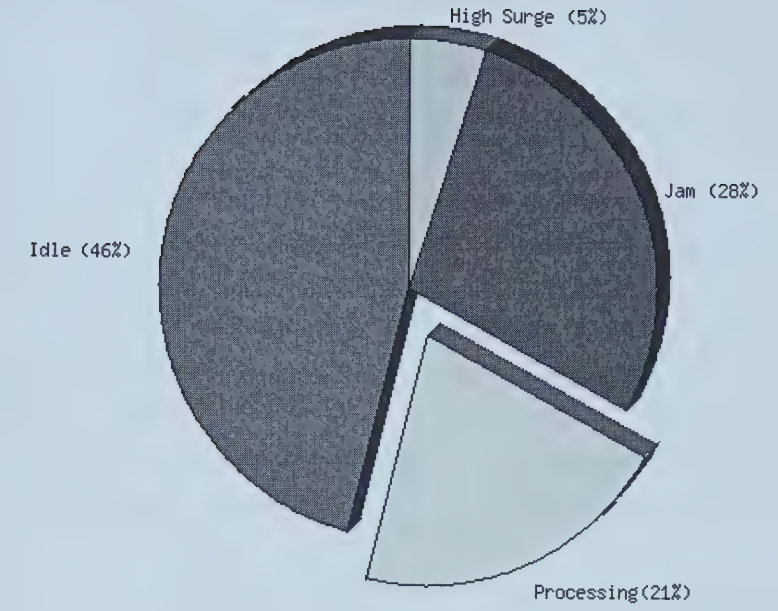
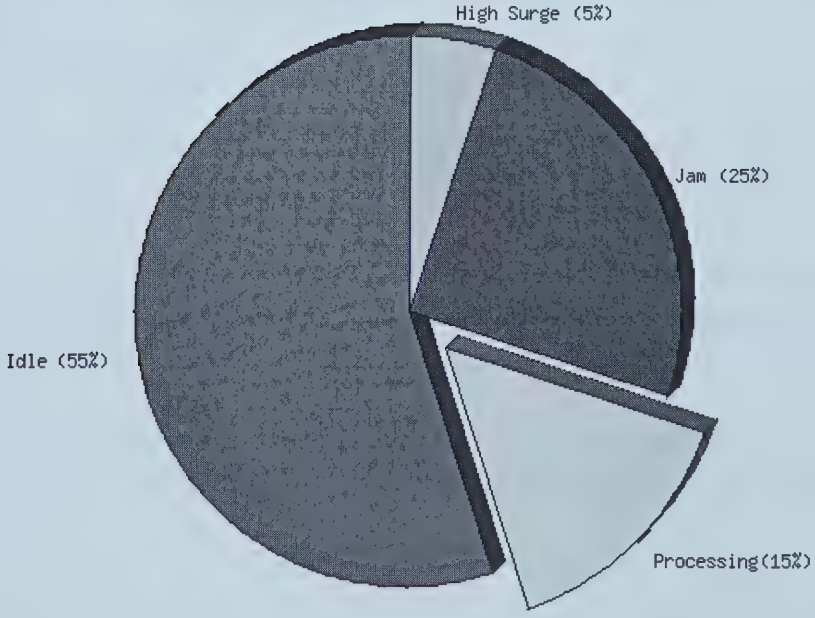


Figure 4.8: Total time within each state, February 2000



(a) January



(b) February

Figure 4.9: Summary of activity durations within `CrusherDown` state

previously assumed.

Earlier we asked the question: what is the average delay following a jam before action is taken to correct the problem? Assuming the operators will always reverse the rolls when a jam is present, our results indicate an average delay of 4 minutes and 40 seconds. This is derived from the 1,082 minutes spent in the `Delay` state, and our record count indicating the `Delay` state was entered 232 times over both months.

One of our focuses is the amount of time where the crusher is actively grinding lumps, the `Processing` state. `Processing` subdivides into `Sure` and `UnSure`, the former being when we are confident of the classification, and the latter being when we are not as sure. Considering these sub-states, we estimate that the lump processing occurred for a total of [5719, 5971] minutes when combining both months. This works out to approximately 4 days, or 7.2% of the operating time.

Of the sub-states belonging to `CrusherDown`, Figure 4.9 shows the relative weight of each. The combination of `Jam`, `Sure`, and `UnSure` reveals that lumps are the cause of 49% of January's down-time and 40% of February's. Our results show that the `Processing` state is active roughly 40% of the downtime credited to lumps.

We need to supply a word of caution at this point. Output precision can only be as accurate as the input available. Given that most of our input points are accurate to the nearest minute, it is possible that a lump lasting 15 seconds (say) would be diagnosed as lasting a minute by our model. As a result, our findings are likely to be somewhat exaggerated.

4.4 Further work

It is known that the rolls draw more current when they are having difficulty grinding material. The model we have presented could be improved by incorporating roll current as another indicator of a lump.

Presently there are no measuring devices at the output of each crusher. Given the amount of time that we have calculated the crushers to be operating within the `Processing` state, it would be interesting to follow-up on this work by comparing the average crusher output for each of the identified states.

There is some curiosity as to whether some oil sand benches are more prone to causing lump problems than others. This analysis could be done by including the source of each oil sand delivery within the model.

4.5 Summary

This chapter serves as an example of how the IAM tool can be used. We built an *Activity Model* for one of Syncrude's double-roll crushers during winter operation. The corresponding *Activity Program* was executed on data points gathered during January and February of 2000.

Two items of interest to Syncrude are:

- The amount of operation time lost as a result of frozen lump incidents
- The total crusher uptime

In response to the first point, we sub-divide the crusher downtime into its constituent causes: frozen lumps (state `Lump`), a lack of upstream-arriving material (state `Idle`), or a downstream station having too much material to process (state `HighSurge`). Merging the data of both months, we find the lack of upstream-arriving material to be responsible for 50% of downtime, frozen lumps causing 45%, and downstream problems only 5%. Our model detected the time spent dealing with lumps to be double what was reported in NMET.

For the second point, we challenge Syncrude's classification of a crusher as being inoperational during lump incidents. Part of a lump duration includes time when the rolls are actively processing the frozen material. Our model offers a method for measuring these lump processing times. Over a two month winter period we found that lump processing time accumulated to roughly 4 days.

Chapter 5

Looking under the hood

This chapter highlights technical issues that arose during our tool development and shows how we have dealt with them.

Section 5.1 summarizes our development platform, SICLE. In Section 5.2 we describe how SICLE was applied towards our implementation. Section 5.3 reveals the database design used to house the output data. A brief chapter review is given in Section 5.4.

5.1 Introduction to SICLE

Our development platform is a unique programming environment titled SICLE[1]. SICLE uses an alternative programming format which tends to cause the programmer to approach the problem differently. This alternative programming format has not yet received widespread acceptance amongst the programming community. A sub-goal of this thesis work was to investigate the utility of SICLE's programming paradigm.

SICLE's base language is Tcl[2], and all of Tcl's functionality is still available to SICLE programs. Although not many new coding constructs are introduced in the SICLE meta-language, the execution style of a SICLE script is considerably different from Tcl. Standard Tcl has a flow of execution that is *procedural*, meaning that the script is executed from beginning to end with branches only for subroutines. In contrast, a SICLE program is *event-driven* as the script initially defines entities and their reactionary behaviour and then waits for one of the anticipated events to occur.

The following subsections survey the core aspects of the SICLE environment. For a complete discussion the reader is referred to Pawel Gburzyński's manual [17].

5.1.1 Object-oriented

SICLE implements its own object-orientedness, requiring only standard Tcl to be installed. A peculiarity of its object implementation is that inheritance is not yet supported.

[1] SICLE is a meta-language created by Professor Pawel Gburzyński of the University of Alberta.
[2] Tcl is a free scripting language pioneered by Professor John Ousterhout of the University of California, Berkeley.

The syntax of SICLE's object-related commands is given in Table 5.1. The `create` and `delete` commands are for object creation and deletion respectively. The `class` command defines new object types, and the `qualify` command assures users that certain properties and behaviours exist among object types.

The `qualify` command warrants more explanation. It does not itself add attributes to an object or alter its behaviour. Rather, it just officially declares that a certain object corresponds to the expectations of the qualifier. The programmer is responsible to ensure that an object being qualified is in fact in compliance with the qualification. For example, SICLE's implementation of mailboxes and interfaces (see Section 5.1.5.1) (a) ensures the necessary behaviours are present and (b) qualifies the objects as being their respective types. The `qualify` command exists to help support special object extensions into a given application or into the SICLE language itself.

5.1.2 Entities

A SICLE entity[3] is a class type whose behaviour resembles a finite state machine (FSM). Multiple SICLE entities can co-exist in parallel; each entity has its own thread of control. For each entity the programmer specifies (1) a number of explicitly defined states, (2) a sequence of commands that execute upon the activation of each state, and (3) the events that will cause a transition into other states. The skeleton syntax of a SICLE entity is shown in Program 5.1.1.

Before an entity can be defined it must first be declared with the `class` command (see Table 5.1). Each entity description is required to have a starting state titled `Start`, with the other state names being user-defined. Each state is linked to a number of command statements, collectively referred to as a *code sequence.* Each code sequence is delimited by curly braces. The transitions are specified as commands within a code sequence, usually at its end.

An instantiation of an entity can be referred to as an *entity instance* or an *entity object.* After executing a code sequence, an entity instance is put to sleep until one of its transitions fires. The entity instance is awakened by the earliest event that triggers a transition. When a transition fires, the other events that were being awaited are cleared and the code sequence of the new state is executed. A code sequence that does not contain transitions will cause the entity object to terminate after its command statements have been executed.

The word *state* now has two connotations. The first is a state in the IAM sense: basic states, cluster states, and orthogonal states. The second is in reference to the current mode of a SICLE entity. To avoid confusion, we will refer to the former state group as "states" and the latter as "modes".

When a SICLE entity enters a new mode, it occasionally requires additional information about the event that triggered the transition. Such information, when available, can be retrieved through

[3]The syntax, as will be seen, uses the keyword `process` and not `entity`. Regardless, we will refer to these structures as entities. This is to avoid confusing the term process with other connotations used previously.

class *typename* [*arglist*] [*constructor*] [*destructor*]

Description	:	Declares a new object type.
typename	:	Following execution of this command, *typename* is a legitimate object type.
arglist	:	Argument list that is passed to the constructor.
constructor	:	Tcl code that is executed on the creation of each object instance.
destructor	:	Tcl code that is executed on the destruction of each object instance.

create *typename arg1 ... argn*

Description	:	Creates a new object instance.
typename	:	The class of the created object.
arg1 .. argn	:	The constructor arguments of the object.

defined qualify *typename qualifier*

Description	:	Checks if a class description has been qualified a certain way.
typename	:	The typename of the class being checked.
qualifier	:	The qualification being checked.

delete [*handle*]

Description	:	Destroys an object instance.
handle	:	The handle to the object being destroyed. If omitted, the object executing the command is destroyed.

implore *handle methodname arg1 ... argn*

Description	:	Same as `invoke`, except that no exception will be triggered if the method *methodname* does not exist.

invoke *handle methodname arg1 ... argn*

Description	:	Attempts to execute the method *methodname*, offered by the *handle* object instance.
handle	:	Handle of the object instance whose method is being accessed.
methodname	:	Name of the method to execute.
arg1 ... argn	:	Parameters passed into the method.

method *typename methodname arglist body*

Description	:	Linking a user-defined function to a class.
typename	:	Identifies the class.
methodname	:	The name of the new function.
arglist	:	The argument list of the new function.
body	:	The body of the new function.

qualify *typename qualifier*

Description	:	Qualifies a class description to be of a certain type. Thereafter the typename can be referenced with confidence that it carries the properties and behaviour expected of the qualification.
typename	:	The name of the class being qualified.
qualifier	:	The qualification given.

Table 5.1: Object-oriented commands

Program 5.1.1 Skeleton syntax of a SICLE entity

```
class basicStateClass { } {
   # initialization commands
}

process basicStateClass {
   # bring data items into scope

   switch $State {
      Start {
         # initialize data items, transfer to the starting user-defined state
      }
      state_2 {
         # a. specify code sequence to execute when state_2 becomes active
         # b. specify transitions into potential next states
      }
      ...
      state_n {
         # same format as state_2
      }
   }
}
```

global variables. The global variable most frequently used for this purpose is titled **Message**. For example, the `attention` command discussed later in Table 5.4 is supported by **Message**.

5.1.3 Transition commands

SICLE's transition commands are listed in Table 5.2.

The `wait` command is the principle command for setting up transitions. The first argument to `wait` is the name of the *Activity Interpreter* (AI) responsible for recognizing when the targeted event has occurred. There are four AI types: a timer AI, a class AI, a mailbox AI, an interface AI. The timer AI triggers timeout alarms; the class AI triggers events when entities enter particular modes, or when they terminate; the mailbox/interface AI trigger events for each change made to them. Mailboxes and interfaces are further described in Section 5.1.5.1.

Each *Activity Interpreter* has a finite number of events that it can respond to. Table 5.3 lists the supported AI types along with their associate events. The leftmost column is the value passed as the 1^{st} parameter to `wait`, the 2^{nd} column is the event passed as the second parameter to `wait`, and the 3^{rd} column is a description of the event. Many of the 3^{rd} column descriptions are taken directly from the SICLE manual [17]. Square brackets in the table indicate that the field does not have a constant value.

The `monitor` command is a variation on the `wait` command. The `wait` command sets up a *one-time* transition; after the transition is followed it is forgotten. In contrast, the `monitor` command is a persistent transition that continuously re-instates itself until either the program terminates or

proceed *nextMode*
Causes an immediate transition into the specified *nextMode*.

wait *ai event [nextMode] [priority]*
Registers a transition to branch to the *nextMode* on the occurrence of the specified *event*. The first argument, the *ai*, refers to the *Activity Interpreter* that is being waited on. *priority* can be set to be `low` or `high`.

monitor *ai event [nextMode] [priority]*
Same as `wait`, except it continually re-registers itself after each transition.

cancel *ai event*
Disables a persistent event previously created with the `monitor` command. *ai* is the *Activity Interpreter* responsible for triggering the persistent event.

Table 5.2: SICLE's transition commands

it is explicitly canceled with a `cancel` command.

Only the first two arguments to `wait` and `monitor` are mandatory. The omission of the `nextMode` parameter specifies the current mode to be re-started. The omission of the `priority` parameter defaults to `low`.

If two transitions are triggered at the same time, the one with the higher priority will fire.

5.1.4 Event interruptions

The specification of asynchronous events implies that we would like to be interrupted as soon as the awaited events occur. In reality, there are often *critical sections* which must be protected from the occurrence of unexpected/inconvenient events. Verifying that all such critical sections have been accounted for, and that the resulting code is robust under all input conditions, can be a very difficult task for complex applications. SICLE provides a programming convenience which curtails the problem almost completely: transitions occur *in-between* the defined modes. The code sequences are non-preemtible. Provided the modes and transitions are defined correctly, this arrangement is a compromise to allow the program execution to be organized and yet still responsive.

To accomplish the above, SICLE defines its own scheduling mechanism. The scheduler decides the order of execution among the entities competing for CPU cycles.

5.1.5 Communication

5.1.5.1 Internal

SICLE allows entities to link to *mailboxes*. A mailbox is simply a repository for data. Any entity can deposit information into a mailbox, but only one entity can read a given mailbox.

An alternate means of interprocess communication is through the `attention` command. A transmitting entity executes the `attention` command as part of a code sequence. The receiving

		The timer AI
$Timer	[delay]	[delay] is the number of milliseconds to delay for.
		The class AI
[object handle]	[mode identifier belonging to the object's class]	Ask for an event to be triggered when the object gets into a particular mode. The event will be triggered *after* the object has completed its processing for that mode.
	Death	Ask an event to be triggered when the object terminates.
		The mailbox AI
[mailbox handle]	NewItem	This event is triggered when an item is deposited into the mailbox.
	Nonempty	This event is triggered when the mailbox is or becomes nonempty.
	Receive	This event is triggered when the mailbox is or becomes nonempty. The first item is automatically extracted from the mailbox; it will be returned in the global variable **Message** when the process is awakened.
	Delete	This event is triggered when an item is removed (extracted) from the mailbox.
	[an integer]	This event is triggered when the number of items in the mailbox becomes (or is) exactly equal to the specified number.
	[anything else]	The parameter is taken to be a regular expression. This event is triggered if the mailbox contains an item matching the expression.
		The interface AI
[interface handle]	NewItem	This event occurs when at least one byte appears in interface's input buffer and is available for extraction The buffer contents must change for the event to occur.
	Receive	This event occurs when the input buffer is or becomes nonempty. The entire buffer contents are returned in global variable **Message** when the event is presented to the waiting process, and the buffer is cleared.
	Close	This event occurs when the interface is closed and the buffer is empty. It will occur immediately after the `wait` request is issued if these conditions are met.
	Client	This event occurs on a server interface when there is a pending connection request. A process receiving this event may execute `client` to accept this connection.
	[an integer]	This event occurs when the length of the string in the input buffer is (or becomes) greater than or equal to the specified number.
	[anything else]	The parameter is taken to be a regular expression. This event is triggered if the contents of the input buffer match the expression.
		All of the above AIs
[any of above]	Attention	This event occurs if some object executes the `attention` command.

Table 5.3: Supported AIs and their events

entity participates by listening for the `Attention` event, and responds to its occurrence. The receiver specifies `Attention` the 2^{nd} argument to a `wait/monitor` command.

The syntax of SICLE's inter-process commands is given in Table 5.4.

attention *ai* [*message*] Creates an `Attention` event on the designated AI. Arbitrary supplemental information can be passed via the *message* parameter, which the target AI can retrieve from the global variable **Message**.
deposit *mailboxHandle item* Inserts a new *item* into the mailbox associated with *mailboxHandle*. Returns 1 if the deposit was successful, or 0 if the mailbox was already full.
extract *mailboxHandle where* [*pattern*] Removes an item from the mailbox pointed to by *mailboxHandle* and stores the removed item in the variable *where*. If the *pattern* parameter is omitted, the first item in the mailbox is removed. Otherwise *pattern* is interpreted as a regular expression, and the first matching item is removed. Returns 1 if an item is successfully extracted from the mailbox, 0 otherwise.

Table 5.4: SICLE inter-process communication commands

5.1.5.2 External

SICLE communicates with external objects through *interfaces*. Examples of external objects include sensors, devices, and sockets. An interface allows all outside objects to be handled uniformly, regardless of their peculiarities. Interfaces have `deposit` and `extract` access commands, similar to mailboxes. These access commands are not described here because interfaces are not part of our IAM implementation. A SICLE program working with interfaces is constrained to the sensors and devices that are supported by the SICLE environment.

5.2 Implementing IAM with SICLE

As our implementation medium, we selected SICLE on the basis of (a) its event-driven execution style and (b) its explicit syntax regarding states and event transitions. In the course of developing our IAM tool, we created a slightly modified version of SICLE tuned for our purpose. These modifications are described as we go along.

SICLE already provides much of the event-driven functionality that we are after. After all, it may not implement Statecharts directly but its code closely resembles that of a finite state machine. Our goal, therefore, was not to implement Statecharts from scratch, but rather to translate a Statecharts specification into the language already understood by SICLE.

5.2.1 Passing information between modes

This subsection details our first SICLE modification: the ability to communicate information between the modes of an entity. By this, we mean having a SICLE mode be able to receive arbitrary data from the previous mode that was active. This can be compared to a function receiving parameters to guide its execution. Although SICLE supports indirect communication through mailboxes, we opted for an approach that tied the information to the transitions themselves.

Consider, for example, a pair of modes that have multiple transition paths between them. Assume that a given mode needs to know which transition fired to activate it. This supplementary knowledge can be passed from the previous mode by attaching the transition name to each of its outgoing transitions.

We implement a modified version of the transition commands: `proceed`, `monitor`, and `wait`. We add an additional parameter that allows extra information to be passed. When a new mode is activated, the extra data can be retrieved by reading the **EventInfo** global variable.

5.2.2 Short-term event memory

New SICLE functions were introduced to keep track of recent event occurrences. The new commands, defined in Table 5.5, do not interfere with the normal SICLE operation. They merely provide a more long-term memory of event occurrences, when this information is needed.

The standard behaviour of SICLE is to automatically erase an entity's event list once a transition is made to a different mode. However, sometimes it is useful to capture all event occurrences and then post-analyse all of the events inside the next mode.

Section 5.2.4 describes how the these new functions eased the implementation of guards.

startMonitoringEvents *entityHandle* Starts collecting the names of all registered events that occur to an entity. Event names are stored in a non-shrinking list. (An event is "registered" once it is subsequently used within a `wait` or `monitor` command).
stopMonitoringEvents *entityHandle* Clears the event list that has been created for the entity. Future events occurring to the identified entity will no longer be stored in a non-shrinking list.
getRecentEvents *entityHandle* Returns the list of recent events accumulated for the specified entity.

Table 5.5: Supplementary event commands

5.2.3 IAM states

This subsection details how we implemented the IAM states.

A separate SICLE class is defined for each state type. Each user-defined state is an object instantiation of its corresponding class. A state object is always in existence, even when it is considered to be inactive. This technique offers a performance boost as we do not require objects to be dynamically created or deleted. Instead, all objects are allocated resources only at program startup.

Before taking a closer look at each IAM state in detail, we shed some light on design aspects shared between the states.

5.2.3.1 Mailbox hierarchy

States need a method of knowing when to become active. We use SICLE's mailboxes to provide this communication medium.

Each cluster and orthogonal state is linked to a separate mailbox. The capacity of each mailbox corresponds to the number of sub-states that can be active at any time. For clusters, the mailbox can hold one data item; for orthogonal states, the mailbox has an entry for each component. Each state object can read and write to the mailbox of its parent, as well as its own mailbox if it has one. A special "root" object, containing just a mailbox, ensures that even top-level states have a parent mailbox.

The mailboxes arrange the state objects into a tree hierarchy. The root mailbox is at the top of the hierarchy, the orthogonal and cluster states form the intermediate levels, and the leaf nodes are the basic states. We will use the term *sub-tree* to refer to the collection of states that descend from a particular cluster/orthogonal state.

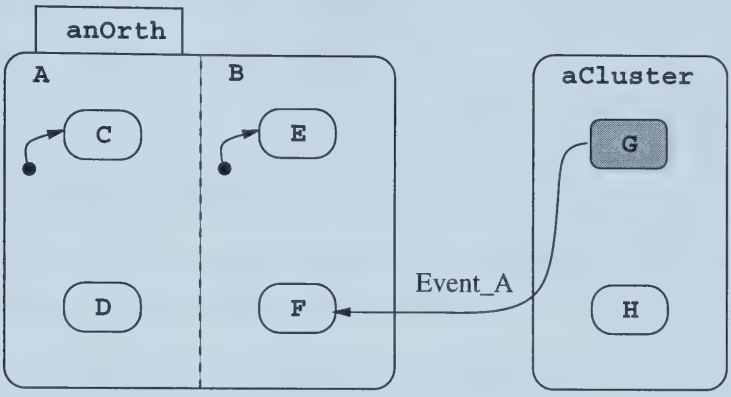
The names of the state(s) to become active next are transferred throughout the mailbox hierarchy. It works as follows. Let π refer to the absolute name of the next active state(s). The name of the starting state(s), π, is initially copied into the root level mailbox. When a state object τ sees its name in its parent mailbox, it activates itself. If π further specifies some descendent(s) of τ, the descendent(s) are activated by having state τ deposit their names in its own mailbox. Otherwise state τ will activate its default sub-state(s), if it has any.

A state object can become inactive in one of three ways: (a) one of its outgoing transitions fires or (b) a transition from an ancestor state fires, or (c) a transition from a descendent fires that points to a state outside the active subtree. When a state object β de-activates itself, it will usually write the name of the next active state in its parent mailbox. The exception to this rule is when the transition from an ancestor state fires (case b), as then sub-state β will not know (nor need to know) the name of the next state to become active.

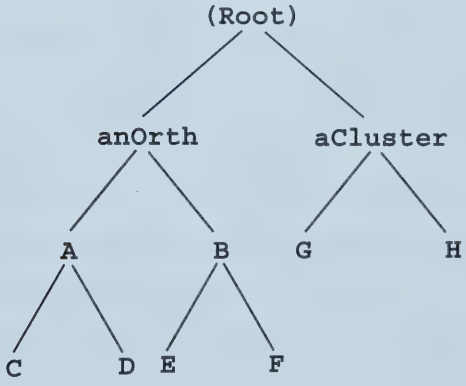
Both clusters and orthogonal states can receive π directives from their sub-states.

An orthogonal state consists of a number of components. Each component is in fact implemented as an instantiation of the cluster class. If an orthogonal state receives a new π directive from one of its components, it immediately de-activates itself and copies the message into its parent mailbox.

A cluster state, upon receiving a new π directive from a sub-state, will first check if the destination is within its subtree. If so, the state ignores π — its sub-states will themselves recognize their names and become active. If not, the state will render itself inactive and copy π into its parent mailbox.



(a) Sample Activity Model



(b) Tree hierarchy

Figure 5.1: Hierarchical breakdown of an activity model

Consider Figure 5.1. There is a sample *Activity Model* drawn, along with the corresponding tree hierarchy. State `G` is darkened to indicate that it is currently active. Assume that *Event_A* occurs and causes a transition into state `F`. The following lists the subsequent mailbox communication that will occur in order to complete the transition:

1. State `G` writes the string ".anOrth.B.F" into the mailbox of state `aCluster`, and then becomes inactive.

2. State `aCluster` responds to its mailbox deposit. After checking that the specified next state

is not one of its sub-states, `aCluster` copies its mailbox contents into the mailbox of the root (".") state. State `aCluster` becomes inactive.

3. State `anOrth` investigates the new deposit in the root state's mailbox. After checking that the specified next state is within its subtree, it activates itself. Then, `anOrth` writes ".anOrth.B.F" and ".anOrth.A.C" into its mailbox. The second string is added because the transition did not indicate a starting state for the `A` component.

4. States `A` and `B` take notice of the new deposits in their parent state. After discovering identification of their sub-states, they each become active. State `A` writes ".anOrth.A.C" into its own mailbox. State `B` writes ".anOrth.B.F" into its own mailbox.

5. States `C` and `F` take notice of the new deposits in their parent states. After finding their own identities in the mailboxes, they each activate themselves.

The above demonstrates that absolute state names, as previously introduced in Section 3.7.7, are passed between mailboxes during a transition.

5.2.3.2 Common properties

There are three properties common to all IAM state types: a link to the parent mailbox, a name, and a priority value. The mailbox and name attribute are used to communicate which state(s) should be active next, as discussed in the previous subsection. The priority attribute ensures super-state transitions are given precedence over sub-state transitions.

As standard SICLE only supports two levels of priority, we enhanced it to support an arbitrary number of priority levels[4]. In our version, each `wait/monitor` command takes a non-negative integer argument. The larger the integer value, the higher the priority.

Every `wait/monitor` command issued by a state object is weighted according to its priority attribute. The priority of a state remains constant throughout execution. Top level states are initialized with a large-numbered priority value. Each sub-state receives priority that is one value lower than that of its parent.

The `proceed` command is treated different, as it is used for an immediate transfer into another mode. The `proceed` command uses a priority level higher than what any state is allowed to have, ensuring that the transfer is not interrupted. If the intention is to allow interruptions, the programmer should use "`wait $Timer 0 $nextState`" instead.

Program 5.2.1 illustrates the SICLE syntax for declaring a state object with these attributes.

5.2.3.3 Basic states

The SICLE modes and possible sequences of execution within a basic state object are drawn in Figure 5.2. The purpose of each identified mode is described in Table 5.6.

[4]We use a maximum priority level of 65535. This constant can be modified in the unlikely event that more priority

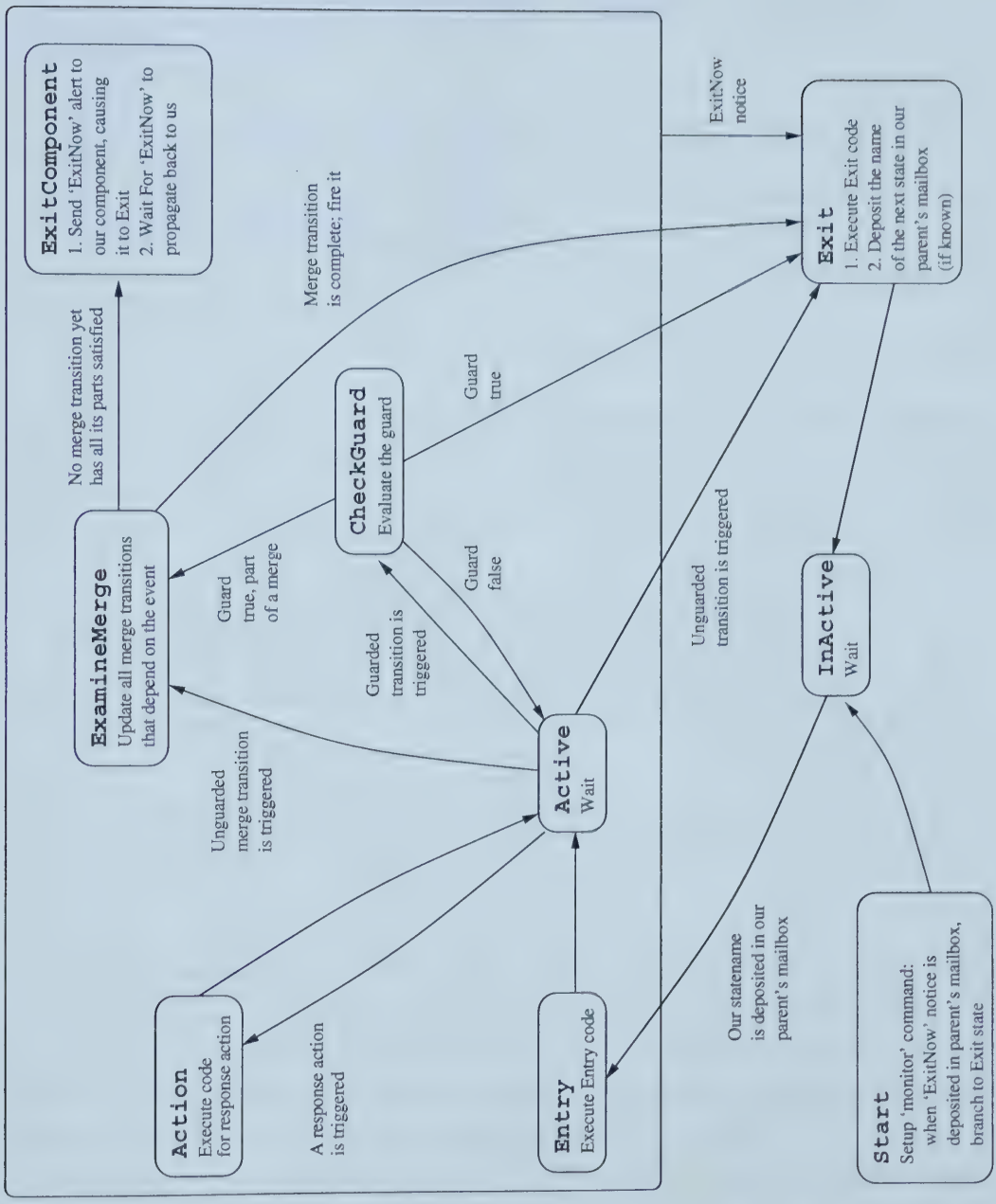


Figure 5.2: Modes of an IAM basic state

Program 5.2.1 Declaring a SICLE object with two parameters

```
class basicStateClass {name priority} {

   # 'useown' is a SICLE command to bring attributes of the current object into scope
   useown MyName MyPriority
   set MyName     $name
   set MyPriority $priority

   # 'StateMbx', 'getParentName' are part of the IAM programming environment
   useown PMailbox    # Points to the mailbox of our parent
   global  StateMbx   # This array is indexed by state, each element is a mailbox
   set ParentStateName [getParentName $MyName]
   set PMailbox $StateMbx($ParentStateName)

   # ... other initialization items
}
```

A special *ExitNow* notice is used when one state wants to tell another to "terminate immediately". It is needed, for example, when a parent state fires an outgoing transition and must first terminate its descendents. Figure 5.2 borrows Statecharts hierarchy notation to indicate that an *ExitNow* notice from a parent state (not included in the figure) will supersede all activities of the basic state.

`Action`	The code for a (non-*Entry*, non-*Exit*) response action is executing.
`Active`	The state exists in the foreground; it is considered active.
`CheckGuard`	A guard associated with an event is being evaluated.
`Entry`	Executes the response action code for the state's *Entry* event.
`Exit`	Executes the response action code for the state's *Exit* event.
`ExamineMerge`	A leg of a merge transition has completed. Note that it is possible that the triggering event is linked to multiple legs, as multiple merge transitions can be defined.
`InActive`	The state exists in the background only; it is not currently active.
`Start`	Obligatory start state. We use it to setup a `monitor` command.

Table 5.6: Modes of a basic state

One thing to notice is that the *Entry* and *Exit* response actions will always execute whenever a state becomes `Active` or `InActive`, respectively. Even if states are entered/exited in subtle ways, we are assured that the *Entry*/*Exit* response actions will not be overlooked. Examples of "subtle" transitions include a cluster state activating its default starting state, or a higher-level state firing an outgoing transition and causing an immediate exit from all its sub-states.

The `ExamineMerge` and `ExitComponent` modes are exclusive to the processing of merge transitions. A merge transition may involve basic sub-states within an orthogonal state. As described in Section 3.2.4, when a portion of a merge transition is satisfied not only is the source state exited but its surrounding component is exited as well. The merge transition will fire when all of the involved components have exited.

levels are needed.

Consider, for example, Figure 5.3. All of the basic states, namely {A, B, C, D, and E}, execute according to the diagram of Figure 5.2. Assume that OrthState is active, meaning that states {A, B, C} are each in their Active mode. Further assume that events will occur in the following order: *e2*, *e1*, *e3*. The occurrence of event *e2* shifts state B into its ExamineMerge mode, where it is realized that none of the outgoing transitions can fire yet. The top transition is awaiting event *e1*; the bottom transition is awaiting event *e3*. State B's mode changes again to ExitComponent where it sends an exit notice to its component. Component_2, upon receiving the notice, disables its active sub-state (state B) and then disables itself. Event *e1* occurs next, moving state A to its ExamineMerge mode. Since the topmost transition is now complete, state A shifts directly to the Exit mode and the merge transition fires. State D will become active next. The event queue is cleared on the firing of the transition. In particular, the occurrence of event *e2* is not remembered the next time state OrthState is re-entered.

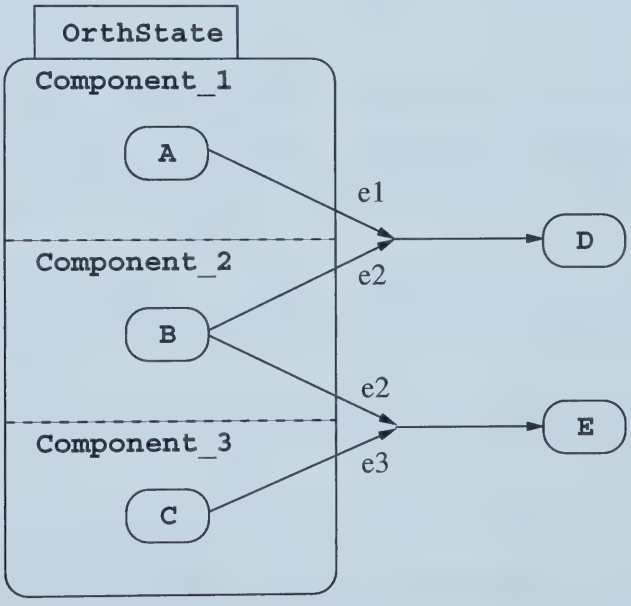


Figure 5.3: Example of a merge transition

SICLE pseudo-code, corresponding to Figure 5.2, is shown in Program 5.2.2. Where the coding details are not instructional they have been replaced with commented descriptions. Parenthesized comments are intended to replace program code; non-parenthesized comments describe the surrounding program code.

5.2.3.4 Cluster states

The internal layout of a cluster state is much the same as it is for the basic state. In fact, a number of the modes call procedures that are shared between the state implementations. This section briefly

Program 5.2.2 SICLE structure of an IAM basic state

```
process basicStateClass {
  useown State MyName MyPriority PMailbox
  switch $State {
    Start {
      # Monitor parent mailbox for "ExitNow" string. When found, enter Exit.
      # The next IAM state is unknown; priority matches that of our parent.
      monitor $PMailbox "ExitNow" Exit "Unknown" [expr $MyPriority + 1]
      proceed InActive
    }
    InActive {
      # Wait until this IAM state is active again, then transfer to Entry mode
      wait $PMailbox "$^{MyName}$" Entry "" $MyPriority
    }
    Entry {
      # (Execute the action code associated with entering this IAM state).
      proceed Active
    }
    Active {
      startMonitoringEvents $This
      # (Execute a 'wait' command for each event that can trigger a response action or
      #  a transition into an outside state).
      # Response actions take us to the Action mode. Guarded transitions take us to
      # the CheckGuard mode. Unguarded merge transitions take us to the
      # ExamineMerge mode. Unguarded non-merge transitions take us to Exit mode.
    }
    ExamineMerge {
      # (Update merge structures. If a merge has completed, goto Exit mode passing
      #  the name of the next IAM state as argument. Otherwise, goto ExitComponent.)
    }
    ExitComponent {
      # Send "ExitNow" signal to our parent component. Then wait for the signal to
      # propagate back to us (our component will inactivate us before exiting itself).
    }
    CheckGuard {
      # Evaluate guard expression for the event that brought us here.
      # If true, goto Exit mode.
      # If false, (A) Evaluate guards for other transitions triggered by the event.
      #           (B) If nothing found, get a list of all the recent events and see if
      #               any other transitions or action codes can fire.
      #           (C) If still nothing found, return to Active once CheckGuard completes.
      stopMonitoringEvents $This    # this clears the event list
    }
    Action {
      # (Execute code for a response action).
      proceed Active
    }
    Exit {
      # (Execute action code associated with exiting this IAM state).
      # (If the next IAM state is known, write it to our parent mailbox).
      proceed InActive
    }
  }
}
```

reviews the additional behaviour that is unique to clusters.

Before a cluster enters its `Exit` mode, it must pass through a `KillSubstate` mode. This mode causes the cluster to send an *ExitNow* notice to its active sub-state. The cluster's `KillSubstate` mode is exited when the child enters its own `InActive` mode. Therefore, when a transition exiting from a cluster fires, its sub-states are immediately exited.

Consider the special case where transitions to outside states are simultaneously fired by a parent and child at approximately the same time. This situation is demonstrated in Figure 5.4. In the figure, state `aCluster` transitions to state `B` on the occurrence of event *e1* and sub-state `A` transitions to state `C` on the occurrence of event *e2*. Assume that events *e1* and *e2* are both recognized at the same time. The parent transition will be executed by SICLE first due to its higher priority. The parent, `aCluster`, will consequently transmit an *ExitNow* notice its active child. The child will now have two events waiting in SICLE's run queue that it must respond to. But the first of these two events is its own transition to an outside state. The problem is that the firing of the child transition conflicts with the firing of the parent transition.

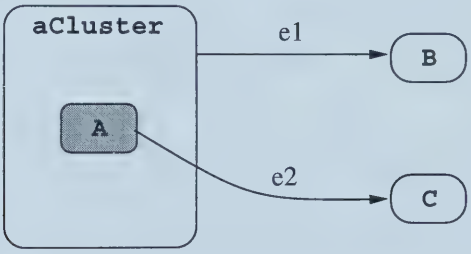


Figure 5.4: Example of a cluster and sub-state each pointing to different next states

We avoid this potential problem by introducing a new SICLE command, described in Table 5.7. The command is used to remove all sub-state events from the run queue, and is executed from within the cluster's `KillSubstate` mode.

ignoreStates *stateList*
This function removes the `wait` commands, whose entity handles are identified in *stateList*, from the run queue. It does not remove the `monitor` commands.

Table 5.7: Supplementary SICLE command: ignoreStates

5.2.3.5 Orthogonal states

A number of minor differences distinguish the implementation of orthogonal states from that of basic states. Program 5.2.3 describes the SICLE structure of an orthogonal state.

As previously mentioned, each component within an orthogonal state is implemented as a cluster object. An orthogonal state must de-activate all of its components before transferring to its

`InActive` mode. This is the purpose of the `KillAllComponents` mode.

For clarity, allow us to point out that `ExitComponent` and `KillAllComponents` are referring to different components. The `ExitComponent` mode is used when our orthogonal state is nested within another orthogonal state, and is part of a merge transition. In this context 'component' refers to the parent of the orthogonal state. In contrast, for the `KillAllComponents` mode each 'component' is a child of the orthogonal state.

5.2.4 IAM transitions

When a user invokes the (IAM) `transition` command, an equivalent (SICLE) `wait` command is stored in an array. The array is indexed by the absolute name of the source state. During program execution, the `Active` mode of each state will dynamically issue all of the `wait` commands associated with the state.

5.2.4.1 Events

Table 5.8 outlines how the event types of Section 3.4 are implemented.

IAM event	SICLE implementation
Timeout	`wait` $Timer <time> ...
OrthEnter	`wait` <handle of state> Entry ...
OrthExit	`wait` <handle of state> Exit ...
ThreshExceeded	`wait` <handle of *Blackboard* variable> <eventName> ...
RangeExceeded	`wait` <handle of *Blackboard* variable> <eventName> ...
Change	`wait` <handle of *Blackboard* variable> Change ...

Table 5.8: Implementation of IAM event types

SICLE is designed to interface directly with sensors, or to simulate sensors. All IAM *Blackboard* variables are simulated sensors within the SICLE environment. Simulated sensors are written-to and read-from the same application. The difference between a simulated sensor and an ordinary variable is that the former can raise (SICLE) events.

SICLE's `attention` command is used to raise an *Attention* event on an entity. We expand this idea to allow any event name, not necessarily *Attention*, to be raised. The IAM threshold event types are implemented by having their user-prescribed name raised on the *Blackboard* variable they are monitoring.

Whenever a **ThreshExceeded** or **RangeExceeded** event is specified with an `eset` command, a callback function is created. The callback function is executed whenever the value of the *Blackboard* variable changes. When the callback function decides that the **ThreshExceeded**/**RangeExceeded** event has occurred, it raises the event on the *Blackboard* variable, which in turn triggers the desired event.

Program 5.2.3 SICLE structure of an IAM orthogonal state

```
process orthStateClass {
  useown State MyName MyPriority PMailbox LocalMailbox
  global AllSubstateHandles
  switch $State {
    ⋮
    # Modes Start, ExamineMerge, ExitComponent, CheckGuard, and Action are
    # omitted. They are the same as for the BasicStateClass.
    ⋮
    InActive {
      # Wait for this IAM state or one of its sub-states to become active again.
      wait $PMailbox "^${MyName}$" Entry "" $MyPriority
      wait $PMailbox "^${MyName}(.)*" Entry "" $MyPriority
    }
    Entry {
      # (Clear history; erase the names of components that have completed their
      #  portion of a merge the last time this IAM state was active).
      # (Execute the action code associated with entering this IAM state).
      # (Activate the sub-states that should be active).
      proceed Active
    }
    Active {
      startMonitoringEvents $This
      # Listen for a change of state announced by one of our sub-states.
      wait $LocalMailbox Newitem ExamineMailbox "" $MyPriority

      # (Execute a 'wait' command for each event that can trigger a response action or
      #  a transition into an outside state).
    }
    ExamineMailbox {
      # ('LocalMailbox' contains the name of the next IAM state to be active; proceed
      #  to Exit).
    }
    Exit {
      # (If any of our components are active...)
          # Ignore pending substate events
          ignoreStates $AllSubstateHandles($MyName)
          # (proceed to KillAllComponents mode).

      # (Otherwise...)
          # (Execute action code associated with exiting this IAM state).
          # (If the next IAM state is known, write it to our parent mailbox).
          proceed InActive
    }
    KillAllComponents {
      # (send 'ExitNow' notice to all components. Once all components have entered
      #  their InActive mode, return to Exit).
    }
  }
}
```

The **Change** event is implemented similarly, except that the name of the event is always *Change* (instead of being user-defined).

5.2.4.2 Guards

Significant to the implementation of guards is that the guard expressions are evaluated *at the moment when the event they are guarding occurs.* Each execution of the `gset` command stores a guard expression for later evaluation.

When a guard is linked to a `transition` command, the next mode of the SICLE entity is `CheckGuard`. Otherwise the next mode is something different. Within the `CheckGuard` mode, the stored guard expression is evaluated.

Let us now assume that the functions described in Section 5.2.2 do not exist. A potential problem would be if two or more of the `Active` mode `wait` commands were fulfilled at the same time, and both `wait` commands were guarded (say). In this case, one of the two events would cause a transfer to the `CheckGuard` mode and the other would be discarded by SICLE. If the guard was false, control would return to `Active` mode without considering the other event(s) that also occurred.

Our solution to this problem, of course, was to introduce the functions of Section 5.2.2. The pseudo-code shown in Program 5.2.2 already illustrates how these functions are used. When a given guard evaluates to false, we call `getRecentEvents` to learn of other events that also occurred.

Without the `getRecentEvents` command, we would have no record of other simultaneous event occurrences and would be forced to wait for them to re-occur again. In cases where input is sparse, a missed event can be a significant issue.

5.2.5 DRS implementation

The *Data Retrieval Server* (DRS) module is not part of our toolkit. We do provide a sample DRS, which we used to compute the Chapter 4 results, but we do not claim this implementation is generic enough for any monitoring application.

Our DRS happens to be coded in C, while the IAM package is built on top of Tcl. A direct consequence is that our IAM package can be ported more easily to other operating systems. A user with an operating system that is not compatible with our DRS implementation can still make use of the IAM package by re-implementing a new DRS module.

Ideally the DRS module can be used to filter the data from noise before it is transmitted to the IAM program. The DRS module is in a better position to do this filtering because

1. It is more closely coupled to the sensors, and presumably understands them better.
2. It can be coded in a faster language than IAM, and therefore have more time to do mathematical processing.

When it is not possible for the DRS to perform filtering, the trending KS can be used as a primitive mechanism to remove noise. This is not guaranteed to be fast, but it may prove useful for short experiments and initial setup.

5.2.6 Communicating with the DRS

IAM has a separate SICLE entity whose responsibility is to communicate with the DRS. We call this entity the *Input Retrieval Unit* (see Figure 3.10 on page 31).

5.2.6.1 Protocol

Communication between the DRS and IAM is done using plain text. The structure of the information passed in either direction is shown in Figure 5.5. All transmission tokens going either direction are delimited with angle brackets.

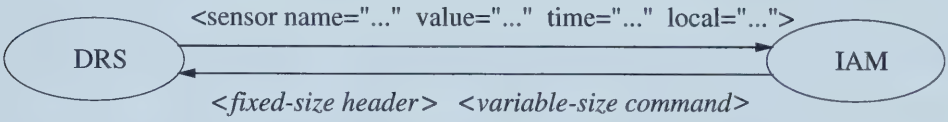


Figure 5.5: Protocol between DRS and IAM

Each command from the IAM to the DRS consists of two tokens. The first token contains a five digit number to indicate the length of the subsequent token. The second transmission can be one of the following commands:

- `register [sensorname]`. This command is issued by the IAM software in response to the user executing a `streamRegister` command.
- `ready`. All memory allocation, and etc, is done by IAM at startup. This command is issued by the IAM software right after it finishes initializing.

The command traveling from the DRS to IAM updates the value of a sensor. The *name* parameter identifies the sensor, *value* is the new value, *time* is the timestamp of the sensor reading, and *local* is the timestamp of the DRS computer. The *local* timestamp is recorded the instant that the DRS computer transmits the token. The next subsection reveals how the *local* timestamp is used to detect cases where the DRS is transmitting data too quickly.

5.2.6.2 Dealing with fast input

If the DRS is coded in a faster programming language, or running on a faster computer, then it will likely transmit data quicker than our IAM program can interpret it.

To ensure this is not happening, each transmission from DRS $\rightarrow$ IAM is time stamped with the *local* time of the DRS machine. The IAM program monitors the relative time difference between its

own clock and the *local* timestamp given in the transmission. If the relative time difference grows beyond a certain threshold[5] then the IAM program aborts on the assumption that it is not keeping up.

5.2.6.3 Scheduling

SICLE internally schedules the CPU between its entities, mailboxes, and interfaces. Mailboxes and interfaces were introduced in Section 5.1.5. Mailboxes serve as a repository for data generated within the SICLE environment. Interfaces serve as a repository for data arriving from external sources (sockets and supported sensor types).

When an interface receives CPU attention, it loads newly arrived data (which has been internally buffered) into the interface buffer. Once data has been loaded into the interface buffer it becomes available to the SICLE entities. In addition, the insertion of the data may generate events awaited by entities.

The idea of SICLE's scheduler it to retrieve a batch of input, process it, and then retrieve another batch, etc. The scheduler works according to the algorithm[6] of Program 5.2.4.

Program 5.2.4 Scheduler within SICLE's kernel

```
while true {
   read input into all interfaces
   while (there exist events to process) {
      schedule CPU among competing entities
   }
}
```

In our IAM implementation all sensor data arrives from the DRS over a socket. The socket is accessed only by the *Input Retrieval Unit* entity via a SICLE interface. The *Input Retrieval Unit* transfers data from the interface to the *Blackboard*, where it is then responded to by all of the IAM state entities. The *Input Retrieval Unit* is assigned a reserved priority level that is higher than what any IAM state is allowed to use, meaning that it has precedence over all of the IAM state entities. The high priority level of the *Input Retrieval Unit* simply ensures that it is the first entity to execute on each iteration through the scheduler's loop.

An unpleasant situation would emerge if it were possible for the DRS to transmit data sufficiently fast that the IAM application only had enough time to read the input without being able to process it. This situation is prevented through two measures. First, the capacity of the interface buffers determine the maximum amount of data that can be loaded into them at any time[7]. Second, the scheduler ensures that all entities (including IAM state entities) have an opportunity to execute before additional data is loaded into the socket interface.

[5] We arbitrarily selected 6 seconds.
[6] This version of the scheduler is a slight modification from the original.
[7] Our implementation allows up to 4 outstanding sensor transmissions in the buffer.

5.3 Database storage

Output from the IAM tool can either be stored as a text file or entered into a local database. The decision is made during installation. When storing to database, the current IAM version will create and write to MySQL[8] tables. Support for other database types is left as future work. The file alternative exists for users that do not have MySQL installed on their machine.

The database is written to as a result of the `record` command (see Section 3.6). By default, `record` saves the list of active states and separately identifies the particular state that invoked the command. The user can supplement the output of `record` by making use of the optional parameters.

We now describe the organization of our database. This knowledge is necessary for users to pose queries on the resultant data.

5.3.1 ER diagram

An Entity-Relationship (ER) model describes data as entities[9], relationships, and attributes [12]. Figure 5.6 reveals the ER diagram we created for storing program output.

The 'OUTPUT' entity contains tuples resulting from the `record` command. It stores a timestamp, the ID of the state that executed the `record` command, the user-defined content and an incrementing integer that serves as the primary key. The microseconds are stored separately within the timestamp because each date/time item is only accurate to the nearest second.

The 'RECORDED_DURING' relationship allows us to look up all of the states that were active at the moment that the `record` command was issued.

The 'STATE' entity translates state IDs into state names and additionally keeps track of each state's parent. 'STATE' participates more than once in the relationship 'PARENT_OF'. In each case role names are used to distinguish the manner that the entity participates.

5.3.2 Database tables

Following the guidelines given by Elmasri and Navathe [12], we mapped the above ER model into database tables (Figure 5.7). The arrows point out the referential integrity constraints.

5.4 Summary

Our implementation explores the use of SICLE, an alternative programming methodology. This chapter summarizes the SICLE language and shows how we applied it towards the development of our IAM application. Some minor shortfalls were discovered; our remedies are described. The chapter finishes by detailing the database structure used to store output.

[8] MySQL is a relational database management system introduced by Michael "Monty" Widenius in 1995. MySQL is open source software.

[9] In the database context, an entity relates to a "thing" – a physical object or conceptual notion. Up to this point, we have referred to an entity in the SICLE context: a programming structure that includes modes and transitions.

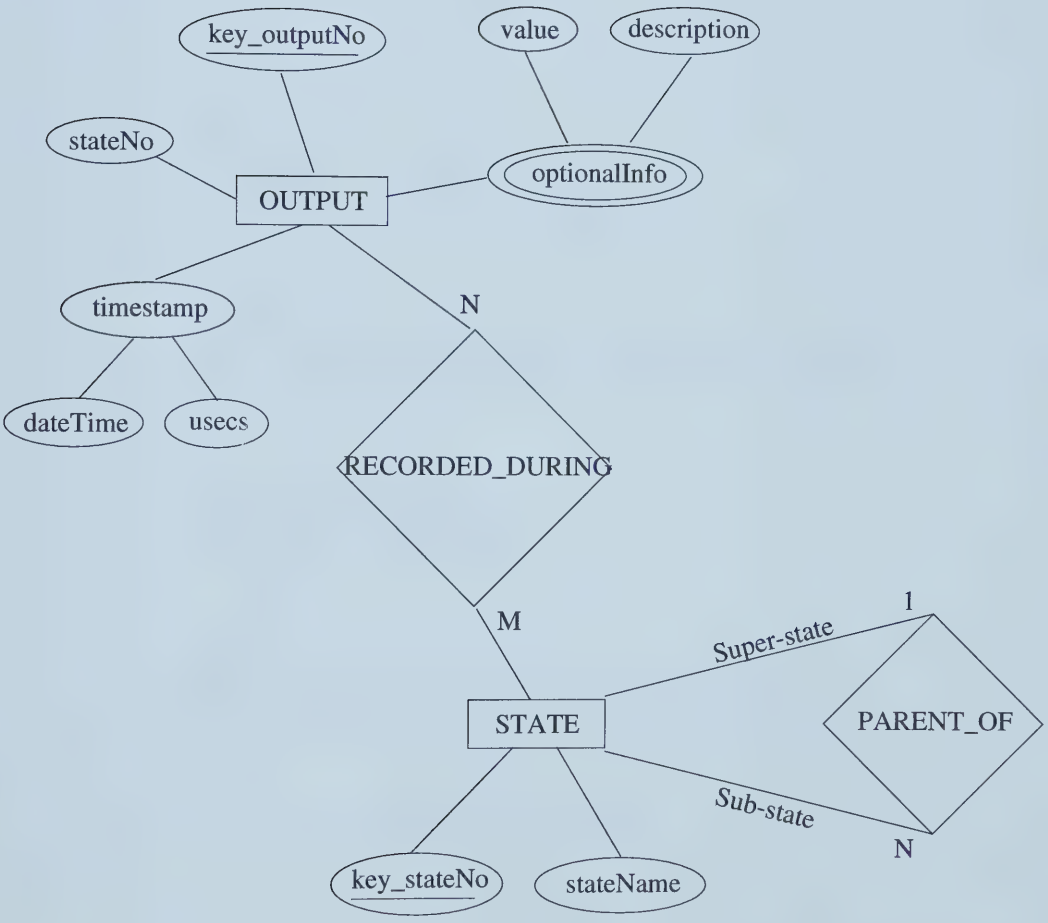


Figure 5.6: Entity-Relationship (ER) model of database organization

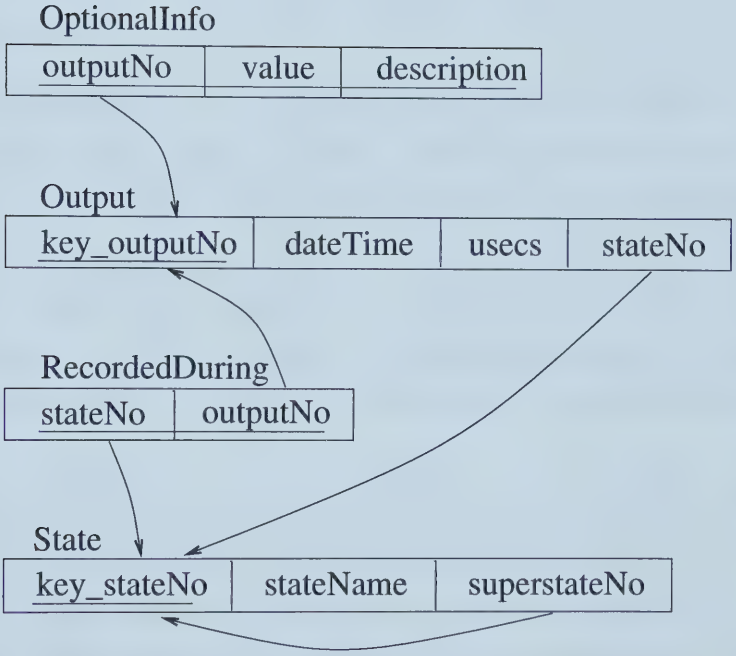


Figure 5.7: Database tables

Chapter 6

Conclusion

6.1 Summary

This thesis has investigated the use of *Activity Models*, a subset of Statecharts, for modelling industrial machines. Their purpose is to understand the context under which other sensor inputs are being recorded. Two example models have been constructed: one for maintaining the disposition of trucks, and one for monitoring the crushing of oil sand lumps during winter operation.

We constructed a prototype tool for executing *Activity Models*. The tool was implemented with SICLE, an event-driven language developed at the University of Alberta. Our prototype allows user-defined response actions to execute from any of the states in the model. Application-specific input sources are decoupled from the tool through the innovation of an intermediary *Data Retrieval Server*.

We executed our crusher model using two months of industrial input data that was made available by Syncrude Canada Ltd. Our *Data Retrieval Server* read through the input and fed it into our prototype tool, properly configured for the application. The results have produced new information for Syncrude on crusher activity, with respect to the processing of frozen oil sand lumps.

The procedure of developing an *Activity Model* to describe some equipment will usually lead to better understanding how the machinery is operated. Further, an *Activity Model* encourages equipment to be optimized by considering operational variables as a function of machine activity.

6.2 Contribution

This thesis has contributed by:

- Reviewing the use of, and proposing changes to, a new methodology for event-driven programming
- Recognizing and motivating the importance of recording contextual data in addition to other machine statistics

- Putting forward the use of Statecharts for specifying the behaviour of equipment, processes, and their surrounding environment
- Implementing a tool that executes according to a subset of the Statecharts specifications
- Demonstrating the practicality of the approach with an industrial example

6.3 Limitations

We only allow response actions to be linked with states, not transitions. This simplification was made on the premise that whatever action was to be associated with a transition could instead be associated with the entrance of the destination state. An unpleasant consequence is that it becomes difficult to learn the frequency with which each transition is followed.

6.4 Possibilities for further research

6.4.1 Graphical user interface

It would be convenient if the activity monitoring could be specified with a graphical user interface (GUI). However, developing a GUI to output executable code can be a challenging task.

Our tool can be used as a middleware language of such a development. A corresponding GUI only needs to output the Statechart specification in the syntax understood by our program. It is our belief that this is simpler than having the GUI program manually implement an entire program from scratch.

6.4.2 Model validation

The ease with which a model can be verified as correct will often decide whether or not such models are constructed in the first place. With our existing toolkit, the principal verification method is an examination of the logged output. As well, the user-defined response actions can be programmed to verify invariant conditions which must hold when certain states are active.

Future work towards this end could include a run-time animation display and/or observer processes. Run-time animation consists of using a GUI to visualize progress through an *Activity Model*, and is related to the discussion of Section 6.4.1.

An observer, in the manner described in [16], would consist of a separate process whose sole purpose is to monitor the run-time execution of the IAM software. An analogy between static assertions and observers is offered by [16]:

> As assertions are typically used to transform the dynamic semantics of a sequential program into a set of static formulas, observers reduce the semantics of a parallel program to the semantics of a sequential program. An observer can be viewed as a dynamic

assertion – a statement about a configuration of [mode] transitions in a collection of [IAM states].

In the context of IAM, each state is a separate thread that is executing in parallel with the other states. Observers would have the ability to execute in response to IAM state changes, and would be given access to all IAM program variables. Their intention would be to monitor execution without interfering with it; observers would not respond to IAM events, nor would they generate any events themselves. They would simply observe the mode transitions that occur within the IAM states. For example, a user may like to assert that the response action of one state always executes prior to the response action of another state. Or perhaps that all of the legs of a merge transition complete within some fixed period of time.

6.4.3 Machine learning

There is a growing popularity for machine-learning principles to discover trends among a large number of data points. We can foresee the use of machine learning algorithms to decide when to transition between states. We have introduced a placeholder (Knowledge Source modules) where machine learning algorithms can be incorporated.

6.4.4 Real-time execution

Some industrial applications require more strict monitoring than what we offer. To satisfy this demand, we see value in a monitoring tool designed for a real-time platform, i.e. Real-Time Linux [5].

One potential application of an activity monitoring program is to disable manufacturing systems immediately if an error state is detected. On this front, a real-time system is likely to be preferred for its guarantees that the shutdown will occur within strict time bounds.

6.4.5 Long-term direction: condition-based maintenance

One long-term application area for *Intelligent Activity Monitoring* is to aid in the development of maintenance diagnosis systems.

Maintenance strategies are a heated research topic for companies using expensive equipment. Machine maintenance not only requires investment in terms of parts and people, but also removes a system from operation until it is fully restored. Maintenance additionally creates the possibility that something will go wrong during the repair process.

Condition Based Maintenance (CBM) can be defined as a maintenance strategy that monitors machinery while it is operating for the purpose of detecting the onset of failure. This contrasts with the more common *Time Directed Maintenance* where equipment is routinely serviced at scheduled times. CBM is gaining interest because it normally does not involve an intrusion into the equipment and the actual preventive action is taken only when it is believed that an incipient failure has been detected [29].

Most approaches to *Condition Based Maintenance* focus on *Condition Monitoring*, defined as a system that tracks the health of a machine or machine component. Our complementary approach is to focus on *Activity Monitoring* whereby the system observes what the machine is doing at any point in time. The idea is that the activity provides contextual data necessary to make sense of the condition monitoring data values.

As an example, response actions defined for the *Entry* and *Exit* events can be useful for CBM. Assuming there is some component whose condition can be sensed directly, a sensor reading can be scheduled at the entry and exit of each state in an *Activity Model*. The difference between these recorded values can indicate how much variability occurred while the state was active. This can lead to an indication of which activities are affecting the component the most, and possibly influence the way a CBM system analyzes the component during those states.

Appendix A

Source Code

Here we present the full syntax for the IAM programs motivated previously. Section A.1 gives the code for the truck torsion tube study of Section 3.8. Section A.2 shows the code for the crusher model described in Chapter 4.

A.1 Truck Model

```
#!/usr/bin/tclsh
package require iam 1.0
iam

INPUT_CONFIG:
  set hostname [info hostname]                    ;# application specific
  set portNo   50000                              ;# application specific
  connectTo $hostname $portNo

  # — raw sensors —
  streamRegister velocity
  streamRegister boxAngle
  streamRegister wheelAngle
  streamRegister truckAngle
  streamRegister weightometer

  # — derived sensors —
  # acceleration
  set inputVariable velocity
  set numArrivals 5
  set historySize 15
  createKS acceleration trend $inputVariable $numArrivals $historySize

  # weightTrend
  set inputVariable weightometer
  set numArrivals 5
  set historySize 15
  createKS weightTrend trend $inputVariable $numArrivals $historySize

  # wheelDirection
  set inputVariable wheelAngle
```

```
  set numArrivals 1
  set divideList   [list 30 60]
  set intervalList [list turningLeft goingStraight turningRight]
  createKS wheelDirection discretization $inputVariable $numArrivals  \
    $divideList $intervalList

CONSTANTS:
  constant MOVE_VELOCITY 5                                    ;# miles per hour
  constant MIN_BOXUP     45                                   ;# degrees
  constant HI_SLOPE      0.07                                 ;# trend slope
  constant LO_SLOPE      0.0                                  ;# trend slope

EVENTS:
  eset Moving              ThreshExceeded (velocity,    >=, $MOVE_VELOCITY, 1)
  eset Box_Raised          ThreshExceeded (boxAngle,    >=, $MIN_BOXUP, 1)
  eset Box_Lowered         ThreshExceeded (boxAngle,    < , $MIN_BOXUP, 1)
  eset Weightometer_Jump   ThreshExceeded (weightTrend, >=, $HI_SLOPE, 1)
  eset Weightometer_Stable ThreshExceeded (weightTrend, <=, $LO_SLOPE, 1)
  eset New_Direction       Change  (wheelDirection)
  eset Model_Stuck         Timeout (04:00:00)

GUARDS:
  gset Box_Up   boxAngle >= $MIN_BOXUP
  gset Box_Down boxAngle <  $MIN_BOXUP

PROCEDURES:
  proc sentAlertEmail { } {
    set who   "trouble@research.com"
    set title "activity model stuck"
    set msg   "stuck in states [getActive]"
    mail $who $title $msg
  }

STATES:
  orth <Truck_Ore_Transport> {
    component <Status> {
      cluster <Loading> {
        event <Entry>       { record         }
        event <Model_Stuck> { sendAlertEmail }

        basic <ReceivingLoad> {
          event <Entry> { record }
        }
        basic <Waiting> {
          event <Entry> {
            set angle [get truckAngle]
            record $angle "Truck angle after load"
          }
        }
      }
      basic <Driving_With_Load> {
        event <Entry>       { record         }
        event <Model_Stuck> { sendAlertEmail }
      }
```

```
      basic <Dumping> {
        event <Entry>       { record         }
        event <Model_Stuck> { sendAlertEmail }
      }
      basic <Driving_With_Box_Raised> {
        event <Entry>       { record         }
        event <Model_Stuck> { sendAlertEmail }
      }
      basic <Driving_Without_Load> {
        event <Entry>       { record         }
        event <Model_Stuck> { sendAlertEmail }
      }
    }
    component <Turning> {
      basic <DirectionMonitor> {
        event <Entry> {
          record [get wheelDirection] "Wheel direction"
        }
      }
    }
  }

TRANSITIONS:
  transition {.Truck_Ore_Transport.Status.Loading.Waiting}            \
             {.Truck_Ore_Transport.Status.Loading.ReceivingLoad}      \
             Weightometer_Jump ""

  transition {.Truck_Ore_Transport.Status.Loading.ReceivingLoad}      \
             {.Truck_Ore_Transport.Status.Loading.Waiting}            \
             Weightometer_Stable ""

  transition {.Truck_Ore_Transport.Status.Loading}                    \
             {.Truck_Ore_Transport.Status.Driving_With_Load}          \
             Moving ""

  transition {.Truck_Ore_Transport.Status.Driving_With_Load}          \
             {.Truck_Ore_Transport.Status.Loading}                    \
             Weightometer_Jump ""

  transition {.Truck_Ore_Transport.Status.Driving_With_Load}          \
             {.Truck_Ore_Transport.Status.Dumping}                    \
             Box_Raised ""

  transition {.Truck_Ore_Transport.Status.Dumping}                    \
             {.Truck_Ore_Transport.Status.Driving_Without_Load}       \
             Moving Box_Down

  transition {.Truck_Ore_Transport.Status.Dumping}                    \
             {.Truck_Ore_Transport.Status.Driving_With_Box_Raised}    \
             Moving Box_Up

  transition {.Truck_Ore_Transport.Status.Driving_With_Box_Raised}    \
             {.Truck_Ore_Transport.Status.Driving_Without_Load}       \
             Box_Lowered ""
```

```
  transition {.Truck_Ore_Transport.Status.Driving_Without_Load}      \
             {.Truck_Ore_Transport.Status.Loading}                   \
             Weightometer_Jump ""
END
```

A.2 Crusher Model

```
#!/usr/bin/tclsh
package require iam 1.0
iam

INPUT_CONFIG:
  set hostname [info hostname]
  set portNo   50000
  connectTo $hostname $portNo

  # — raw sensors —
  streamRegister C08_Reverse
  streamRegister C08_Forward
  streamRegister Maintenance
  streamRegister LumpSignal
  streamRegister HopperLvl
  streamRegister RollSpeedC08
  streamRegister ApronFeeder
  streamRegister PileLevel12
  streamRegister PileLevel13

  # — derived sensors: none —

CONSTANTS:
  constant rollsThresh  90                                          ;# 90 RPM
  constant apronThresh  30                                          ;# 30 %
  constant hopperThresh 50                                          ;# 50 %
  constant surgeThresh  75                                          ;# 75 %

EVENTS:
  eset forwardRolls  Change (C08_Forward)
  eset reverseRolls  Change (C08_Reverse)
  eset rollsOff      ThreshExceeded (RollSpeedC08, < , $rollsThresh, 1)
  eset rollsOn       ThreshExceeded (RollSpeedC08, >=, $rollsThresh, 1)
  eset apronOn       ThreshExceeded (ApronFeeder,  > , $apronThresh, 1)
  eset apronOff      ThreshExceeded (ApronFeeder,  <=, $apronThresh, 1)
  eset lumpSignalOff ThreshExceeded (LumpSignal,   <=, 0, 1)
  eset lumpSignalOn  ThreshExceeded (LumpSignal,   > , 0, 1)
  eset startMaint    ThreshExceeded (Maintenance,  > , 0, 1)
  eset endMaint      ThreshExceeded (Maintenance,  <=, 0, 1)
  eset hiHopper      ThreshExceeded (HopperLvl,    >=, $hopperThresh, 1)

GUARDS:
  gset HiHopper   HopperLvl >= $hopperThresh
  gset LowHopper  HopperLvl <  $hopperThresh
  gset HiSurge    ((PileLevel12 >= $surgeThresh)                    \
```

```
                  OR (PileLevel13 >= $surgeThresh))                \
                  AND (HopperLvl >= $hopperThresh)
  gset NoReason   ((PileLevel12 < $surgeThresh)                    \
                  AND (PileLevel13 < $surgeThresh))                \
                  AND (HopperLvl >= $hopperThresh)
  gset NoOverride NOT (IN(.Operating.IAM.CrusherDown.Lump.Jam.Unjam.Rev))
  gset RollsOn    RollSpeedC08 >= $rollsThresh
  gset RollsOff   RollSpeedC08 <  $rollsThresh
  gset NoReasonRollsOff          (NoReason) AND (RollsOff)
  gset NoReasonRollsOnLowHopper ((NoReason) AND (RollsOn)) AND (LowHopper)
  gset NoReasonRollsOnHiHopper  ((NoReason) AND (RollsOn)) AND (HiHopper)
  gset RollsOnHiHopper          (RollsOn) AND (HiHopper)
  gset RollsOnLowHopper         (RollsOn) AND (LowHopper)
  gset NoOverrideHiHopper       (NoOverride) AND (HiHopper)
  gset NoOverrideLowHopper      (NoOverride) AND (LowHopper)

PROCEDURES:

STATES:
  orth <Operating> {
    event <Entry> { record "" "Operating Entry" }
    event <Exit>  { record "" "Operating Exit"  }

    component <NMET> {
      event <Entry> { record "" "NMET Entry" }
      event <Exit>  { record "" "NMET Exit"  }

      basic <R_NoLump> {
        event <Entry> { record "" "R_NoLump Entry" }
        event <Exit>  { record "" "R_NoLump Exit"  }
      }
      basic <R_Lump> {
        event <Entry> { record "" "R_Lump Entry" }
        event <Exit>  { record "" "R_Lump Exit"  }
      }
    }
    component <IAM> {
      event <Entry> { record "" "IAM Entry" }
      event <Exit>  { record "" "IAM Exit"  }

      basic <CrusherUp> {
        event <Entry> { record "" "CrusherUp Entry" }
        event <Exit>  { record "" "CrusherUp Exit"  }
      }
      cluster <CrusherDown> {
        event <Entry> { record "" "CrusherDown Entry" }
        event <Exit>  { record "" "CrusherDown Exit"  }

        cluster <NoLump> {
          event <Entry> { record "" "NoLump Entry" }
          event <Exit>  { record "" "NoLump Exit"  }

          basic <Idle> {
            event <Entry> { record "" "Idle Entry" }
```

```
            event <Exit>  { record "" "Idle Exit"  }
          }
          basic <HighSurge> {
            event <Entry> { record "" "HighSurge Entry" }
            event <Exit>  { record "" "HighSurge Exit"  }
          }
        }
        cluster <Lump> {
          event <Entry> { record "" "Lump Entry" }
          event <Exit>  { record "" "Lump Exit"  }

          cluster <Processing> {
            event <Entry> { record "" "Processing Entry" }
            event <Exit>  { record "" "Processing Exit"  }

            basic <Sure> {
              event <Entry> { record "" "Sure Entry" }
              event <Exit>  { record "" "Sure Exit"  }
            }
            basic <UnSure> {
              event <Entry> { record "" "UnSure Entry" }
              event <Exit>  { record "" "UnSure Exit"  }
            }
          }
          cluster <Jam> {
            event <Entry> { record "" "Jam Entry" }
            event <Exit>  { record "" "Jam Exit"  }

            basic <Delay> {
              event <Entry> { record "" "Delay Entry" }
              event <Exit>  { record "" "Delay Exit"  }
            }
            cluster <Unjam> {
              event <Entry> { record "" "Unjam Entry" }
              event <Exit>  { record "" "Unjam Exit"  }

              basic <Rev> {
                event <Entry> { record "" "Rev Entry" }
                event <Exit>  { record "" "Rev Exit"  }
              }
              basic <Fwd> {
                event <Entry> { record "" "Fwd Entry" }
                event <Exit>  { record "" "Fwd Exit"  }
              }
            }
          }
        }
      }
    }
  }
}
basic <SchedMaint> {
  event <Entry> { record "" "SchedMaint Entry" }
  event <Exit>  { record "" "SchedMaint Exit"  }
}
```

```
TRANSITIONS:
  transition {.Operating}  {.SchedMaint} startMaint ""
  transition {.SchedMaint} {.Operating}  endMaint   ""

  transition {.Operating.IAM.CrusherDown} {.Operating.IAM.CrusherUp}  \
             apronOn NoOverride

  transition {.Operating.IAM.CrusherUp}                               \
             {.Operating.IAM.CrusherDown.Lump.Processing.Sure}        \
             apronOff NoReasonRollsOnHiHopper

  transition {.Operating.IAM.CrusherUp}                               \
             {.Operating.IAM.CrusherDown.Lump.Processing.UnSure}      \
             apronOff NoReasonRollsOnLowHopper

  transition {.Operating.IAM.CrusherUp}                               \
             {.Operating.IAM.CrusherDown.Lump.Jam}                    \
             apronOff NoReasonRollsOff

  transition {.Operating.IAM.CrusherUp}                               \
             {.Operating.IAM.CrusherDown.NoLump.Idle}                 \
             apronOff LowHopper

  transition {.Operating.IAM.CrusherUp}                               \
             {.Operating.IAM.CrusherDown.NoLump.HighSurge}            \
             apronOff HiSurge

  transition {.Operating.IAM.CrusherUp}                               \
             {.Operating.IAM.CrusherDown.Lump.Jam.Unjam}              \
             reverseRolls ""

  transition {.Operating.NMET.R_Lump}                                 \
             {.Operating.NMET.R_NoLump}                               \
             lumpSignalOff ""

  transition {.Operating.NMET.R_NoLump}                               \
             {.Operating.NMET.R_Lump}                                 \
             lumpSignalOn ""

  transition {.Operating.IAM.CrusherDown.Lump.Processing}             \
             {.Operating.IAM.CrusherDown.Lump.Jam}                    \
             rollsOff ""

  transition {.Operating.IAM.CrusherDown.Lump.Processing}             \
             {.Operating.IAM.CrusherDown.Lump.Jam.Unjam}              \
             reverseRolls ""

  transition {.Operating.IAM.CrusherDown.Lump.Processing.UnSure}      \
             {.Operating.IAM.CrusherDown.Lump.Processing.Sure}        \
             hiHopper ""

  transition {.Operating.IAM.CrusherDown.Lump.Jam}                    \
             {.Operating.IAM.CrusherDown.Lump.Processing.Sure}        \
```

```
            rollsOn NoOverrideHiHopper

  transition {.Operating.IAM.CrusherDown.Lump.Jam}                    \
             {.Operating.IAM.CrusherDown.Lump.Processing.UnSure}      \
             rollsOn NoOverrideLowHopper

  transition {.Operating.IAM.CrusherDown.Lump.Jam.Delay}              \
             {.Operating.IAM.CrusherDown.Lump.Jam.Unjam}              \
             reverseRolls ""

  transition {.Operating.IAM.CrusherDown.Lump.Jam.Unjam.Rev}          \
             {.Operating.IAM.CrusherDown.Lump.Jam.Unjam.Fwd}          \
             forwardRolls ""

  transition {.Operating.IAM.CrusherDown.Lump.Jam.Unjam.Fwd}          \
             {.Operating.IAM.CrusherDown.Lump.Jam.Unjam.Rev}          \
             reverseRolls ""

  transition {.Operating.IAM.CrusherDown.NoLump}                      \
             {.Operating.IAM.CrusherDown.Lump.Jam.Unjam}              \
             reverseRolls ""

  transition {.Operating.IAM.CrusherDown.NoLump.Idle}                 \
             {.Operating.IAM.CrusherDown.Lump}                        \
             hiHopper ""

END
```

Bibliography

[1] *FOLDOC: free on-line dictionary of computing.* http://www.foldoc.org/.

[2] *Merriam-Webster's WWWebster Dictionary.* http://www.m-w.com/.

[3] *Syncrude Canada Ltd.* http://www.syncrude.com/.

[4] K. Aminian, Ph. Robert, E. E. Buchser, B. Rutschmann, D. Hayoz, and M. Depairon. Physical activity monitoring based on accelerometry: validation and comparison with video observation. *Medical Biological Engineering Computing*, 37(3):304–8, May 1999.

[5] Michael Barabanov. A linux-based real-time operating system. Master's thesis, New Mexico Institute of Mining and Technology, 1997.

[6] Bernard T. Barcio. Smooches: State machines for object-oriented concurrent, hierarchical engineering specifications. Master's thesis, University of Texas at Austin, December 1994.

[7] Vadim V. Bulitko and David C. Wilkins. Using petri nets to represent context in blackboard scheduling. In *Proceedings of the AAAI Workshop on Reasoning in Context for AI Applications*, July 1999.

[8] Lionel Cayer. Crusher questions. Syncrude email to J. Gamble, October 26 2000.

[9] Lionel Cayer. Crusher threshold question. Syncrude email to J. Gamble, October 17 2000.

[10] Steve Cook and John Daniels. *Designing Object Systems: Object-Oriented Modelling with Syntropy.* Prentice Hall, Hemel Hempstead, Hertfordshire, 1994.

[11] Daniel D. Corkill. Blackboard systems. *AI Expert*, 6(9):40–47, September 1991.

[12] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, Inc., Redwood City, CA 94065, second edition, 1994.

[13] Tom Fawcett and Foster Provost. Activity monitoring: Noticing interesting changes in behavior. In *Proceedings of the fifth ACM SIGKDD international conference on knoweledge discovery and data mining*, August 1999.

[14] Chris G. Fowler and Sylvia Gonzalez. Caterpillar 793B torsion tube torque study. Research Department Progress Report 28(3), Syncrude Canada Limited, Edmonton, Alberta, March 1999.

[15] Justin Gamble and C. Ronald Kube. Towards a model for intelligent activity monitoring. Research Department Progress Report 29(9), Syncrude Canada Limited, Edmonton, Alberta, September 2000.

[16] Pawel Gburzyński. *Protocol Design for Local and Metropolitan Area Networks.* Prentice Hall, Englewood Cliffs, New Jersey, 1996.

[17] Pawel Gburzyński. *The SICLE Control Package.* appHome, Inc., Sacramento, CA, 1.0 edition, 1999.

[18] W.E.L Grimson, C. Stauffer, R. Romano, and L. Lee. Using adaptive tracking to classify and monitor activities in a site. In *Proceedings of the 1998 IEEE Conference on Computer Vision and Pattern Recognition*, June 1998.

[19] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[20] David Harel and Amnon Naamad. The STATEMATE semantics of statecharts. *ACM Transactions on software engineering and methodology*, 5(4):293–333, October 1996. (Preliminary version appeared as Technical Report, i-Logix, Inc., 1989).

[21] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts.* McGraw-Hill, New York, 1998.

[22] Nancy G. Leveson, Mats P.E. Heimdahl, Holly Hildreth, and Jon D. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.

[23] Paul Jay Lucas. An object-oriented language system for implementing concurrent, hierarchical, finite state machines. Master's thesis, University of Illinois at Urbana-Champaign, 1993.

[24] Tadao Murata. Petri nets: Properties, analysis, and applications. *Proceedings of the IEEE*, 77(4):541–580, April 1989.

[25] Jörg Niere and Albert Zündorf. Testing and simulating production control systems using the fujaba environment. In *Proceedings of AGTIVE Workshop '99, Applications of Graph Transformation with Industrial Relevance*, September 1999.

[26] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, Bonn: Institut für Instrumentelle Mathematik, 1962. Also, English translation, "Communication with Automata." New York: Griffiss Air Force Base. Tech. Rep. RADC-TR-65-377, vol. 1, Suppl. 1, 1966.

[27] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[28] Michael A. Swartwout, Christopher A. Kitts, and Rajesh K. Batra. Persistence-based production rules for on-board satellite automation. In *Proceedings of the 1999 IEEE Aerospace Conference*, March 1999.

[29] Albert H.C. Tsang. Condition-based maintenance: tools and decision making. *Journal of Quality in Maintenance Engineering*, 1(3):3–17, 1995.

[30] W.M.P. van der Aalst. Putting high-level petri nets to work in industry. *Computers in Industry*, 25(1):45–54, 1994.

[31] W.M.P. van der Aalst and M.A. Odijk. Analysis of railway stations by means of interval timed coloured petri nets. *Real-Time Systems*, 9(3):241–263, 1995.

[32] Michael von der Beeck. A comparison of statecharts variants. 863:128–148, September 1994.

[33] Dirk Wodtke, Jeanine Weissenfels, Gerhard Weikum, and Angelika Kotz Dittrich. The mentor project: Steps towards enterprise-wide workflow management. In *Proceedings of the International Conference on Data Engineering (ICDE)*, 1996.

[34] MengChu Zhou and Kurapati Venkatesh. *Modeling, Simulation, and Control of Flexible Manufacturing Systems: A Petri Net Approach.* World Scientific, 1999.